Longest Common Subsequence in Sublinear Space[☆]

Masashi Kiyomi[a,*], Takashi Horiyama[b], Yota Otachi[c]

[a]*School of Data Science, Yokohama City University*
[b]*Faculty of Information Science and Technology, Hokkaido University*
[c]*Department of Mathematical Informatics, Nagoya University*

Abstract

We present the first o(n)-space polynomial-time algorithm for computing the length of a longest common subsequence. Given two strings of length n, the algorithm runs in O(n^3) time with O$\left(\frac{n \log^{1.5} n}{2^{\sqrt{\log n}}}\right)$ bits of space.

Keywords: longest common subsequence, space-efficient algorithm

1. Introduction

A *subsequence* of a string is a string that can be obtained from the original string by removing some elements. If two strings X and Y contain a string Z as their subsequences, then Z is a *common subsequence* of X and Y. For example, "tokyo" and "kyoto" have "to" and "oo" as common subsequences but not "too." The problem of finding (the length of) a *longest common subsequence* is a classic problem in computer science. The applications of the problem range over many fields (see [10, 11, 3, 5] and the references therein). Given two strings of length n, the problem can be solved in O(n^2) time with O($n \log n$) bits of space using a dynamic programming approach [14, 9]. On the other hand, it is known that under the strong exponential time hypothesis, there is no O($n^{2-\varepsilon}$)-time algorithm for any $\varepsilon > 0$ [4, 1, 6, 2]. Given this lower bound, subquadratic-time approximation algorithms have been proposed (see [8, 12] and the references therein). Recently, Cheng et al. [7] presented an approximation algorithm of factor $1 - o(1)$ that runs in polynomial time with polylogarithmic space.

In this paper, we seek for *exact* (i.e., non-approximation) polynomial-time algorithms with small space complexity. To the best of authors' knowledge, there was no known polynomial-time algorithm that runs with o(n) bits of space. A natural (but probably quite challenging) question in this direction would be whether there is a polynomial-time algorithm that runs with *truly* sublinear space of O($n^{1-\varepsilon}$) bits for some $\varepsilon > 0$ [7]. We make a step toward this goal by giving a polynomial-time algorithm that runs with *slightly* sublinear space. More precisely, the result of this paper is as follows.

Theorem 1.1. *Given two strings of length n, the length of a longest common subsequence can be computed in* O(n^3) *time with* O$\left(\frac{n \log^{1.5} n}{2^{\sqrt{\log n}}}\right)$ *bits of space.*

2. The algorithm

We use the standard computational model in the literature of space-efficient algorithms. That is, we take the RAM model with the following restrictions:

- the input is in a *read-only* memory;
- the output must be produced on a *write-only* memory;
- an additional memory that is *readable and writable* can be used.

We measure space consumption in the number of bits used within the additional memory. Throughout the paper, we fix the base of logarithms to 2. That is, $\log x$ means $\log_2 x$.

For nonnegative integers m and n, we denote by $\Gamma_{m,n}$ the directed acyclic graph such that

$$V(\Gamma_{m,n}) = \{v_{i,j} \mid 0 \le i \le m,\, 0 \le j \le n\},$$
$$E(\Gamma_{m,n}) = \{(v_{i,j}, v_{i+1,j}) \mid 0 \le i \le m-1,\, 0 \le j \le n\} \cup$$
$$\{(v_{i,j}, v_{i,j+1}) \mid 0 \le i \le m,\, 0 \le j \le n-1\} \cup$$
$$\{(v_{i,j}, v_{i+1,j+1}) \mid 0 \le i \le m-1,\, 0 \le j \le n-1\}.$$

Namely, $\Gamma_{m,n}$ is the $(m+1) \times (n+1)$ grid with the main diagonal edge in each square, where each edge is oriented from left to right and from top to bottom. See Figure 1.
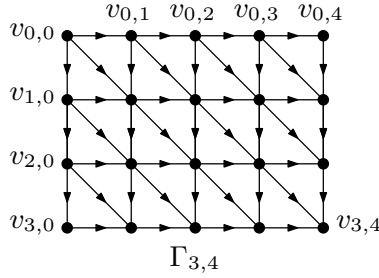

Figure 1: The directed acyclic graph $\Gamma_{m,n}$ with $m = 3$ and $n = 4$.

Let $S = s_1 s_2 \cdots s_m$ and $T = t_1 t_2 \cdots t_n$ be strings of length m and n, respectively. By $\Gamma(S,T)$, we denote the graph $\Gamma_{m,n}$ with the edge weights defined as follows: the horizontal and vertical edges have weight 0; the diagonal edge $(v_{i,j}, v_{i+1,j+1})$ has weight 1 if $s_{i+1} = t_{j+1}$; otherwise it has weight 0. It is easy to see that every $v_{0,0}$–$v_{|S|,|T|}$ path is monotone in both x and y directions and the positive weight edges in the path represent a common subsequence of S and T. Moreover, we can show the following fact, which is used in most of the existing algorithms.

Observation 2.1 (Folklore). The length of a longest common subsequence of S and T is equal to the longest path length from $v_{0,0}$ to $v_{|S|,|T|}$ in $\Gamma(S,T)$.

As mentioned in the introduction, the length of a longest common subsequence of S and T, or equivalently the longest path length from $v_{0,0}$ to $v_{|S|,|T|}$ in $\Gamma(S,T)$, can be computed in $\mathrm{O}(mn)$ time with $\mathrm{O}(m \log n)$ bits of space, where $m < n$. We describe the idea of this algorithm in a slightly generalized setting.

We denote by $\Gamma_{m,n}^w$ the graph $\Gamma_{m,n}$ with an edge weighting w, where each edge weight can be represented in $\mathrm{O}(\log n)$ bits. Let $\lambda_w(v_{i,j}, v_{i',j'})$ be the longest path length from $v_{i,j}$ to $v_{i',j'}$

in $\Gamma_{m,n}^w$. Then λ_w can be expressed in a recursive way as follows:

$$\lambda_w(v_{0,0}, v_{i,j}) = \begin{cases} 0 & i = j = 0, \\ \lambda_w(v_{0,0}, v_{0,j-1}) + w(v_{0,j-1}, v_{0,j}) & i = 0, \ j \geq 1, \\ \lambda_w(v_{0,0}, v_{i-1,0}) + w(v_{i-1,0}, v_{i,0}) & i \geq 1, \ j = 0, \\ \max \left\{ \begin{array}{l} \lambda_w(v_{0,0}, v_{i-1,j}) + w(v_{i-1,j}, v_{i,j}), \\ \lambda_w(v_{0,0}, v_{i,j-1}) + w(v_{i,j-1}, v_{i,j}), \\ \lambda_w(v_{0,0}, v_{i-1,j-1}) + w(v_{i-1,j-1}, v_{i,j}) \end{array} \right\} & i \geq 1, \ j \geq 1. \end{cases}$$

Clearly, we can compute $\lambda_w(v_{0,0}, v_{m,n})$ in $\mathrm{O}(mn)$ time. While a naive implementation takes $\mathrm{O}(mn \log n)$ bits of space, by a simple trick of computing the entries for $v_{i,j}$ in an increasing order of j and storing only two recent rows (that is, the rows $j-1$ and j), we can reduce the space consumption to $\mathrm{O}(m \log n)$ bits. We call this recursive method the *standard algorithm*.

We now explain the high-level idea of our algorithm. Observe that the problem of deciding whether $\lambda_w(v_{0,0}, v_{m,n}) \geq \ell$ is in NL since we can nondeterministically find the next vertex in a longest path in logspace and we can forget the visited vertices as the graph is acyclic. Hence, Savitch's theorem [13] implies that the problem can be solved deterministically with $\mathrm{O}(\log^2 n)$ bits of space but in quasipolynomial $(n^{\mathrm{O}(\log n)})$ time. Such an algorithm recursively find the center of a longest path, and thus has a search tree of maximum degree n and depth $\log n$. Our algorithm can be seen as a modification of this algorithm. Instead of guessing a single vertex included in a longest path, we guess a set of vertices that a longest path intersects. The search tree of our algorithm has maximum degree $\mathrm{O}(2^{\sqrt{\log n}})$ and depth $\mathrm{O}(\sqrt{\log n})$. This gives us a sublinear-space polynomial-time algorithm.

We now prove the main lemma based on the idea described above, which immediately implies Theorem 1.1.

Lemma 2.2. *For an edge weighting $w \colon E(\Gamma_{m,n}) \to \mathbb{Z} \cup \{-\infty\}$ given as a constant time oracle, where each edge weight can be represented in $\mathrm{O}(\log n)$ bits, the longest path length from $v_{0,0}$ to $v_{m,n}$ in $\Gamma_{m,n}^w$ can be computed in $\mathrm{O}(n^3)$ time using $\mathrm{O}\!\left(\frac{n \log^{1.5} n}{2^{\sqrt{\log n}}}\right)$ bits of space.*

PROOF. Without loss of generality assume that $m \leq n$. We also assume that m is a power of 2 by adding, if necessary, some dummy columns and rows and set the weights of the new edges to $-\infty$. The numbers of columns and rows are doubled in the worst case. Note that we do not have to construct the dummy columns and rows explicitly. We just need to remember the original m and n with $\mathrm{O}(\log n)$ bits of additional space consumption.

Let $B = \lceil (n+1)/2^{\sqrt{\log n}} \rceil$. We compute the length in a recursive way. In the recursive algorithm described below, m and n may get smaller in each recursive call, while we keep B the same. The algorithm solves a slightly generalized problem, where we compute the longest path lengths from $v_{0,0}$ to the vertices in $T := \{v_{m,j} \mid (\lceil (n+1)/B \rceil - 1)B \leq j \leq n\}$.

The algorithm. Our algorithm solves the problem in a recursive way. (See Algorithm 1 for the outline.) If $\min\{m, n\} \leq 2B$, then we solve the problem with the standard algorithm.

Assume that $m, n > 2B$. Let $V_h = \{v_{m/2,j} \mid hB \leq j \leq \min\{(h+1)B - 1, n\}\}$ for $0 \leq h \leq \lceil (n+1)/B \rceil - 1$. Observe that for every $u \in T$, each $v_{0,0}$–u path intersects V_h for some h. (Note that such h is not necessarily unique.) Such a path first goes through the upper-left part induced by $\{v_{i,j} \mid 0 \leq i \leq m/2, \ 0 \leq j \leq \min\{(h+1)B - 1, n\}\}$, reaches V_h, then goes through the bottom-right part induced by $\{v_{i,j} \mid m/2 \leq i \leq m, \ hB \leq j \leq n\}$, and finally reaches u. See Figure 2.
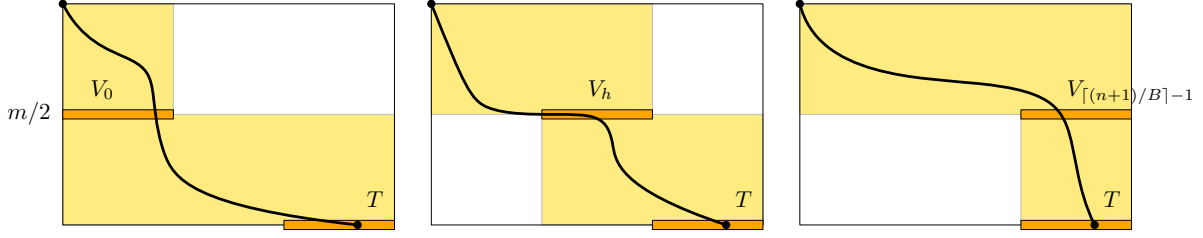
Figure 2: Every path intersecting V_h goes through upper-left and bottom-right regions of the grid.

Based on the observation above, we divide the problem into $\lceil (n+1)/B \rceil$ pairs of subproblems as follows. Let $h \in \{0, \ldots, \lceil (n+1)/B \rceil - 1\}$. We first find the longest path lengths from $v_{0,0}$ to the vertices of V_h in $\Gamma_{m,n}^w$. We find the lengths by recursively applying the algorithm to $\Gamma_{m/2,\min\{(h+1)B-1,n\}}^{w'}$, where w' is obtained from w by restricting it to $\Gamma_{m/2,\min\{(h+1)B-1,n\}}$. The recursive call returns the longest path lengths $\boldsymbol{\ell}'$ from $v_{0,0}$ to V_h, where $\boldsymbol{\ell}'$ is a $|V_h|$-dimensional vector indexed by the second coordinates $hB, hB+1, \ldots, \min\{(h+1)B-1, n\}$ of the vertices.

Let w'' be the edge weighting of $\Gamma_{m/2,n-hB}$ obtained from w and $\boldsymbol{\ell}'$ as follows:

$$w''(v_{0,j}, v_{0,j+1}) = \boldsymbol{\ell}'(j + hB + 1) - \boldsymbol{\ell}'(j + hB) \qquad \text{for } 0 \leq j \leq |V_h| - 2,$$
$$w''(v_{i,j}, v_{i',j'}) = w(v_{i+m/2,j+hB}, v_{i'+m/2,j'+hB}) \qquad \text{otherwise.}$$

Namely, w'' is obtained from w by first restricting it into its bottom-right part starting at V_h, and then changing the weight of the edges in V_h with respect to the longest path lengths from $v_{0,0}$. We now recursively apply the algorithm to $\Gamma_{m/2,n-hB}^{w''}$ for computing $\boldsymbol{\ell}''$, the longest path lengths from V_h to T under w''.

Now we set $\boldsymbol{\ell}(j) = \max_{0 \leq h \leq \lceil (n+1)/B \rceil - 1} \boldsymbol{\ell}'(hB) + \boldsymbol{\ell}''(j - hB)$ for $(\lceil (n+1)/B \rceil - 1)B \leq j \leq n$. The correctness of this final step follows from the claim below.

Claim 2.3. The maximum length of a path P from $v_{0,0}$ to $v_{m,j} \in T$ passing through V_h is $\boldsymbol{\ell}'(hB) + \boldsymbol{\ell}''(j - hB)$.

PROOF (OF CLAIM 2.3). Let k be the maximum index such that $v_{m/2,k} \in V_h$ is included in P. Then, the length of P is $\lambda_w(v_{0,0}, v_{m/2,k}) + \lambda_w(v_{m/2,k}, v_{m,j})$.

Let Q be the subpath of P from $v_{m/2,k}$ to $v_{m,j}$. The length of Q is $\lambda_w(v_{m/2,k}, v_{m,j})$. Let P'' be the $v_{0,0}$–$v_{m/2,j-hB}$ path in $\Gamma_{m/2,n-hB}^{w''}$ that first takes the unique $v_{0,0}$–$v_{0,k-hB}$ path of length $\sum_{0 \leq p \leq k-hB-1}(\boldsymbol{\ell}'(p + hB + 1) - \boldsymbol{\ell}'(p + hB)) = \boldsymbol{\ell}'(k) - \boldsymbol{\ell}'(hB)$ and then follows Q by shifting each vertex coordinate by $-(m/2, hB)$. Since $\boldsymbol{\ell}'(k) = \lambda_w(v_{0,0}, v_{m/2,k})$, the length of P'' is $\lambda_w(v_{0,0}, v_{m/2,k}) + \lambda_w(v_{m/2,k}, v_{m,j}) - \boldsymbol{\ell}'(hB)$. Since P'' is a $v_{0,0}$–$v_{m/2,j-hB}$ path in $\Gamma_{m/2,n-hB}^{w''}$, it holds that

$$\boldsymbol{\ell}''(j - hB) \geq \lambda_w(v_{0,0}, v_{m/2,k}) + \lambda_w(v_{m/2,k}, v_{m,j}) - \boldsymbol{\ell}'(hB). \tag{1}$$

Let Q'' be a $v_{0,0}$–$v_{m/2,j-hB}$ path of length $\boldsymbol{\ell}''(j - hB)$ in $\Gamma_{m/2,n-hB}^{w''}$. Let q be the maximum index such that $q \leq |V_h| - 1$ and $v_{0,q}$ is included in Q''. The unique $v_{0,0}$–$v_{0,q}$ path in $\Gamma_{m/2,n-hB}^{w''}$ has length $\sum_{0 \leq p \leq q-1}(\boldsymbol{\ell}'(p + hB + 1) - \boldsymbol{\ell}'(p + hB)) = \boldsymbol{\ell}'(q + hB) - \boldsymbol{\ell}'(hB)$. From the construction of w'', the rest of P'' starting at $v_{0,q}$ has length $\lambda_{w''}(v_{0,q}, v_{m/2,j-hB}) = \lambda_w(v_{m/2,q+hB}, v_{m,j})$. Since $\boldsymbol{\ell}'(q + hB) = \lambda_w(v_{0,0}, v_{m/2,q+hB})$,

$$\boldsymbol{\ell}''(j - hB) = \lambda_w(v_{0,0}, v_{m/2,q+hB}) + \lambda_w(v_{m/2,q+hB}, v_{m,j}) - \boldsymbol{\ell}'(hB)$$
$$\leq \lambda_w(v_{0,0}, v_{m/2,k}) + \lambda_w(v_{m/2,k}, v_{m,j}) - \boldsymbol{\ell}'(hB). \tag{2}$$

Equations (1) and (2) imply that $\boldsymbol{\ell}''(j - hB) + \boldsymbol{\ell}'(hB) = \lambda_w(v_{0,0}, v_{m/2,k}) + \lambda_w(v_{m/2,k}, v_{m,j})$. ◇

4

Algorithm 1 The algorithm given in the proof of Lemma 2.2.

1: **procedure** LONGESTPATHLENGTH($\Gamma_{m,n}^{w}$, B)
2:      **if** $\min\{m,n\} \leq 2B$ **then**
3:          Use the standard algorithm and return the longest path lengths.
4:      **else**
5:          $\boldsymbol{\ell} := \mathbf{0}$ $\triangleright \ \lambda_w(v_{0,0}, v_{m,j})$ for $(\lceil (n+1)/B \rceil - 1)B \leq j \leq n$
6:          **for** $h \in \{0, \ldots, \lceil (n+1)/B \rceil - 1\}$ **do**
7:              $\boldsymbol{\ell}' := $ LONGESTPATHLENGTH($\Gamma_{m/2,\min\{(h+1)B-1,n\}}^{w'}$, B)
8:              Compute w'' from w and $\boldsymbol{\ell}'$.
9:              $\boldsymbol{\ell}'' := $ LONGESTPATHLENGTH($\Gamma_{m/2,n-hB}^{w''}$, B)
10:              **for** $j \in \{(\lceil (n+1)/B \rceil - 1)B, \ldots, n\}$ **do**
11:                  $\boldsymbol{\ell}(j) := \max\{\boldsymbol{\ell}(j), \boldsymbol{\ell}'(hB) + \boldsymbol{\ell}''(j-hB)\}$
12:          **return** $\boldsymbol{\ell}$

Space consumption. In the recursion tree of Algorithm 1, each inner node stores $\mathrm{O}(B \log n)$ bits for $\boldsymbol{\ell}$, $\boldsymbol{\ell}'$, and $\boldsymbol{\ell}''$. The weight functions w' and w'' are also stored at each inner node and provided as constant time oracles for the children in the recursive tree. Observe that w' can be represented by w with a constant number of indices, and w'' can be represented by w with a constant number of indices and $\boldsymbol{\ell}'$. In total, an inner node stores $\mathrm{O}(B \log n)$ bits of information. Each leaf node executes the standard algorithm with $\min\{m,n\} \leq 2B$, and thus only needs $\mathrm{O}(B \log n)$ bits. Since the depth of the recursion tree is $\lceil \log(m/(2B)) \rceil < \sqrt{\log n}$, the total space consumption is $\mathrm{O}(B \log n \cdot \sqrt{\log n}) = \mathrm{O}(n \log^{1.5} n / 2^{\sqrt{\log n}})$.

Running time. We estimate an upper bound $L(m)$ of the number of leaves in the recursion tree, where $m+1$ is the number of rows. If $m \leq 2B$, then $L(m) \leq 1$. Assume that $m > 2B$. Then

$$L(m) \leq 2 \lceil (n+1)/B \rceil \cdot L(m/2) < (2(2^{\sqrt{\log n}} + 1))^{\sqrt{\log n}}$$

since the algorithm invokes at most $2\lceil (n+1)/B \rceil \leq 2^{\sqrt{\log n}} + 1$ recursive calls with the parameter $m/2$ and the depth of recursion is at most $\lceil \log(m/(2B)) \rceil < \sqrt{\log n}$. Since $(2^{\sqrt{\log n}} + 1)^{\sqrt{\log n}} \leq e \cdot (2^{\sqrt{\log n}})^{\sqrt{\log n}} \leq en$, it holds that $L(m) \in \mathrm{O}(2^{\sqrt{\log n}} \cdot n)$. Since each inner node of the recursion tree has two or more children, the number of all nodes in the recursion tree is also $\mathrm{O}(2^{\sqrt{\log n}} \cdot n)$. Each leaf node takes $\mathrm{O}(Bn)$ time, and each inner node takes $\mathrm{O}(n)$ time excluding the time spent by its children. Therefore, the total running time is $\mathrm{O}(Bn \cdot 2^{\sqrt{\log n}} \cdot n) = \mathrm{O}(n^3)$. □

3. Concluding Remarks

We have presented an algorithm for computing the length of a longest common subsequence of two string of length n in $\mathrm{O}(n^3)$ time using $\mathrm{O}\left(\frac{n \log^{1.5} n}{2^{\sqrt{\log n}}}\right)$ bits of space. The challenge for finding a polynomial-time algorithm with $\mathrm{O}(n^{1-\varepsilon})$ space for some constant $\varepsilon > 0$ remains unsettled.

Our algorithm in Lemma 2.2 is designed for a slightly general problem on the edge weighted grid-like graph $\Gamma_{m,n}^{w}$. The generality allows us to compute some other similarity measures of strings as well. For example, the *edit distance* (or the *Levenshtein distance*) between two strings is the minimum number of operations (insertions, deletions, or substitutions) required to make the strings the same. We can formulate this problem as the shortest path problem on $\Gamma_{m,n}^{w}$ even in a general setting where each operation has different cost possibly depending on the symbols involved. Since the shortest path problem on $\Gamma_{m,n}^{w}$ is equivalent to the longest path problem on $\Gamma_{m,n}^{-w}$, Lemma 2.2 implies that we can compute the (general) edit distance in the same time and space complexity.

References

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *FOCS 2015*, pages 59–78, 2015. `doi:10.1109/FOCS.2015.14`.

[2] 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 *STOC 2016*, pages 375–388, 2016. `doi:10.1145/2897518.2897653`.

[3] Alberto Apostolico. String editing and longest common subsequences. In *Handbook of Formal Languages, Volume 2. Linear Modeling: Background and Application*, pages 361–398. 1997. `doi:10.1007/978-3-662-07675-0_8`.

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

[5] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *SPIRE 2000*, pages 39–48, 2000. `doi:10.1109/SPIRE.2000.878178`.

[6] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *FOCS 2015*, pages 79–97, 2015. `doi:10.1109/FOCS.2015.15`.

[7] Kuan Cheng, Zhengzhong Jin, Xin Li, and Yu Zheng. Space efficient deterministic approximation of string measures. *CoRR*, abs/2002.08498, 2020. `arXiv:2002.08498`.

[8] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the $\sqrt{n}$ barrier. In *SODA 2019*, pages 1181–1200, 2019. `doi:10.1137/1.9781611975482.72`.

[9] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975. `doi:10.1145/360825.360861`.

[10] Daniel S. Hirschberg. Recent results on the complexity of common subsequence problems. In *Time Warps, String Edits, and Macromolecules*, pages 323–328. 1983.

[11] Mike Paterson and Vlado Dancík. Longest common subsequences. In *MFCS 1994*, volume 841 of *Lecture Notes in Computer Science*, pages 127–142, 1994. `doi:10.1007/3-540-58338-6_63`.

[12] Aviad Rubinstein and Zhao Song. Reducing approximate longest common subsequence to approximate edit distance. In *SODA 2020*, pages 1591–1600, 2020. `doi:10.1137/1.9781611975994.98`.

[13] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. `doi:10.1016/S0022-0000(70)80006-X`.

[14] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974. `doi:10.1145/321796.321811`.