Simple, deterministic, fast (but weak) approximations to edit distance and Dyck edit distance

Michal Koucký*[1] and Michael Saks†[2]

[1]Computer Science Institute of Charles University, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
[2]Department of Mathematics, Rutgers University, Piscataway, NJ, USA

Abstract

We consider the problem of obtaining approximation algorithms for standard edit distance and Dyck edit distance that are simple, deterministic and fast, but whose approximation factor may be high. For the standard edit distance of two strings, we introduce a class of simple and fast algorithms called *basic single pass algorithms*. Saha (2014) gave a randomized algorithm in this class that achieves an $O(d)$ approximation on inputs x, y whose edit distance is $O(d)$. In this paper, we (1) present a deterministic algorithm in this class that achieves similar performance and (2) prove that no algorithm (even randomized) in this class can give a better approximation factor. For the Dyck edit distance problem, Saha gave a randomized reduction from Dyck edit distance to standard two string edit distance at a cost of a $O(\log d)$ factor where d is the Dyck edit distance. We give a deterministic reduction whose description and proof are very simple.

*Email: koucky@iuuk.mff.cuni.cz. Partially supported by the Grant Agency of the Czech Republic under the grant agreement no. 19-27871X.

†Email: msaks30@gmail.com. Supported in part by Simons Foundation under award 332622.

1 Introduction

1.1 Background

Edit distance is a measure of similarity of strings. The edit distance of two strings x and y is the minimum number of insertions, deletions and substitutions one has to perform on x to obtain y. This has many applications, from text-processing to bio-informatics.

The classical dynamic programming algorithm to compute edit distance of two strings of length n runs in time $O(n^2)$ [34] and despite much effort there has been only limited success in improving its running time [24, 30]. Indeed, it has been shown that improving its running time substantially would contradict the Strong Exponential Time Hypothesis (SETH) [2, 4, 9, 19]. Hence, there is considerable interest in designing faster algorithms for approximating the edit distance [1, 6, 8, 11–13,28]. Recently, there has been major progress in that regard [5,7,15–17,20,27] leading to a constant factor approximation algorithm for edit distance running in time $O(n^{1+\epsilon})$ for any fixed $\epsilon > 0$ [7]. Despite its near perfect asymptotic running time its inherent complexity renders it impractical. On top of that the approximation factor grows doubly-exponential with $1/\epsilon$. For practical purposes much simpler algorithms would be needed.

A very simple algorithm for approximating edit distance was designed by Saha [31] (and used as an ingredient to obtain fast approximation algorithms to the Dyck edit distance problem; more on this problem below). Saha's algorithm matches symbols from x and y greedily from left to right and whenever it encounters a mismatch, flips a coin to choose which of the two mismatched symbols to delete. The number of deleted symbols is obviously an upper bound on the edit distance d, and it can be shown that with high probability, the number of deleted symbols is $O(d^2)$. While this is a rather weak approximation, for small values of d it provides a reasonably good estimate.

A process similar to Saha's algorithm was implicitly used in randomized embeddings of edit distance into Hamming distance of Chakraborty et al. [21]. This embedding has distortion d with good probability and its analysis boils down to the same random walk process on a line as Saha's algorithm. The embedding on Chakraborty et al. serves also as a basis for sketching edit distance with best sketches of size $O(d^3)$ [14, 25]. The sketch size is directly related to the distortion of the embedding. Practicality of the embedding was also explored for computing the join operation [35]. There are further uses of the process with applications for sublinear-time algorithms [26].

Saha's algorithm can be placed in the framework of *single pass algorithms*. We make a single pass through each string, maintaining a pointer to a "current symbol" in each. At each step we compare the current symbols. If they match then advance both pointers. If they don't match then select one of the symbols to delete and advance that pointer. In Saha's algorithm, the selection decisions are made uniformly at random with no memory of the previous process, but one could allow the selection decision to depend on some past information.

This leads to two fundamental questions: (1) Can Saha's process be derandomized: is there a deterministic selection rule that achieves a similar approximation? and (2) Are there selection rules (deterministic or randomized) that can give a better approximation? In this paper, we provide a positive answer to the first question and a partial negative answer to the second.

1.2 Our results

This paper has three main results.

A simple deterministic single pass algorithm with $O(d)$-approximation factor. In Section 5 we provide a simple deterministic rule which determines which mismatching symbol to remove to obtain an $O(d)$-factor estimate of the edit distance. The rule depends on the past in a very simple way: the choice of which symbol to delete depends only on the number of symbols deleted so far. We call such a rule an *basic selection rule*. Single pass algorithms with basic selection rule can be implemented to run in $O(\log n)$ space.

Limits on approximation quality of single pass algorithms. One might speculate that a different randomized rule could provide a better approximation factor. For example consider a process where we pick a random positive integer i with probability $O(1/i \log^2 i)$, and then during next i mismatches we remove the mismatching symbols all either from x or from y. Such a process seems to do well when the mismatches are located near each other in the strings. This rule is an example of a randomized basic selection rule, i.e., for each choice of the random coins, the selection decision only depends on the number of deletions done previously. In Section 6 we show that no randomized basic selection rule can give approximation factor better than $\Omega(d)$, making our deterministic procedure essentially optimal. We do not know whether a selection rule that depends on more than the number of previous deletions can do better.

A simple deterministic reduction from Dyck edit distance approximation to standard edit distance approximation. We also consider the more general problem of approximating Dyck edit distance. The Dyck language is the language over strings of left and right parentheses, consisting of those strings that can be reduced to the empty string by repeatedly removing a matching pair consisting of a left parenthesis followed immediately by a right parenthesis. In the generalized Dyck language there are multiple types of left and right parentheses and a matching pair consists of a left parenthesis followed by a right parenthesis of the same type. The Dyck edit distance problem is to determine the minimum number of symbol insertions, deletions or substitutions to put the string of typed parentheses in the generalized Dyck language. This is an important problem for error recovery in parsing structured texts and for other applications. It is a generalization of the standard edit distance of two strings; there is an equivalence between the edit distance of two strings, and Dyck edit distance of LR-strings, which are strings consisting of a single block of left parentheses followed by a single block of right parentheses. Saha [31] found an efficient reduction from Dyck edit distance to edit distance, but the reduction and its proof are quite involved. One ingredient in the reduction is the greedy randomized algorithm for ordinary edit distance mentioned earlier. We originally attempted to make the reduction deterministic by replacing the greedy randomized algorithm by our deterministic alternative. In doing this, we realized that there was a further simplification that avoids using either of these algorithms, and yields a deterministic reduction that has similar (slightly better) time and approximation performance to Saha's and is both very simple and has a very simple proof. This is presented in Section 7.

Our deterministic reduction can be combined with our deterministic approximation algorithm for the ordinary edit distance to give a deterministic linear-time approximation algorithm for Dyck edit distance whose approximation factor can be expressed as $O(d \log d)$ or $O(\sqrt{n} \log(n))$.

2 Related work

Edit distance can be computed exactly in time $O(n^2)$ using dynamic programming [34]. In close to 50 years this was improved only slightly: Masek and Paterson [30] gave $O(n^2/\log n)$ time algorithm, and the current asymptotically fastest algorithm by Grabowski [24] runs in time $O(n^2 \log\log n/\log^2 n)$.

Landau et el. [28] gave an algorithm that on strings of edit distance d runs in time $O(n + d^2)$; for large d this is again quadratic. There is an indication that the running time cannot be improved substantially: Backurs and Indyk [9] showed that an algorithm for edit distance running in time $O(n^{2-\delta})$ for some $\delta > 0$ would contradict the Strong Exponential Time Hypothesis (SETH). Abboud et al. [4] showed that even shaving an arbitrarily large polylog factor from n^2 would have the plausible, but apparently hard-to-prove, consequence that NEXP does not have non-uniform NC^1 circuits. Further obstacles for progress are shown in [2, 19].

In the past two decades there has been a lot of progress in designing fast approximation algorithms. An $\sqrt{n}$-approximation algorithm for edit distance running in linear time can be directly obtained from the $O(n + d^2)$-time algorithm of Landau et el. [28]. However, this algorithm relies on longest common extension queries which require large space and sophisticated data structures to implement them efficiently. Bar-Yossef et al. [11] improved the approximation factor to $n^{3/7}$ using an algorithm running in quasi-linear time. Subsequently Batu et al. [13] designed quasi-linear time algorithm with approximation factor $n^{1/3+o(1)}$ which was further improved by Andoni and Onak to $2^{\widetilde{O}(\sqrt{\log n})}$ in near-linear time [8], and to $(\log n)^{O(1/\epsilon)}$, for every $\epsilon > 0$ where the algorithm runs in $n^{1+\epsilon}$ time [6]. Eventually, in the past five years, there was a sequence of works culminating in constant-factor approximation randomized algorithm running in near-linear time $n^{1+\epsilon}$ [5, 7, 15–17, 20, 27]. All these algorithms are quite involved and all of them use space at least $n^{\Omega(1)}$. Previous to this paper, the only small space algorithm for approximating edit distance that we are aware of is the randomized algorithm of Saha described in the introduction, which uses logarithmic space and gives $\sqrt{n}$-approximation.

Prior to our work there was no nontrivial deterministic polynomial-time algorithm for approximating edit distance using sub-polynomial space.

For Dyck edit distance, the dynamic programming algorithm takes time $O(n^3)$. Using fast matrix multiplication Bringmann et al. [18] solved the exact Dyck edit distance in sub-cubic time $O(n^{2.824})$. Abboud et al. [3, 29, 32] show that algorithms for Dyck edit distance running in time faster than matrix multiplication time $O(n^\omega)$ imply surprising algorithms for the k-Clique problem. Using fast matrix multiplication Saha [32] designed $(1 + \epsilon)$-approximation algorithm for Dyck edit distance running in time $O(n^\omega)$. Recently, the running time was improved to $O(n^2)$ by Das et al. [22] using the recent techniques for the ordinary edit distance. Using those techniques, Das et al. also obtains $O(1)$-approximation algorithm for Dyck edit distance running in time $O(n^{1.971})$.

Furthermore, for any $k > 1$, Saha [33] gives $O(n^2 k)$-time approximation algorithm for Dyck edit distance with *additive* approximation $O(n/k)$. Similarly to the algorithm of Landau et el., there are algorithms for Dyck edit distance d running in time $O(n + d^{4.78})$ [10, 22, 23].

In combination with the state of the art ordinary edit distance algorithms, Saha's [31] reduction of Dyck edit distance to ordinary edit distance gives randomized $O(\log n)$-approximation algorithm for Dyck edit distance running in time $O(n^{1+\epsilon})$.

Given the complexity involved in any of these algorithms save Saha's reduction, it would be a stretch to call any of them practical. Our simple deterministic reduction for Dyck edit distance combined with our simple deterministic algorithm for ordinary edit distance could be considered practical; the only drawback is its approximation factor $O(d \log d)$. For some applications this could be acceptable.

3 Basic definitions

Let Σ be a fixed alphabet. For $x \in \Sigma^*$ (the set of finite strings over Σ), the length of x is denoted $|x|$, and the index set of x, denoted $\mathbf{IND}_x$, is $\{1, \ldots, |x|\}$. For $i \in \mathbf{IND}_x$, $x[i]$ denotes the symbol in position i. We frequently will be considering two strings x, y at the same time and we want to treat i as an x-index as distinct from i as a y-index. When there is ambiguity we will write i_x or i_y to indicate which string is being indexed. Similarly, for a subset A of integers we write A_x to mean that A is viewed as a subset of $\mathbf{IND}_x$.

We put the usual product order on $\mathbf{IND}_x \times \mathbf{IND}_y$: $(i,j) \leq (i',j')$ if $i \leq i'$ and $j \leq j'$; and $(i,j) < (i',j')$ if $(i,j) \leq (i',j')$ and $(i,j) \neq (i',j')$. We also write $(i,j) \ll (i',j')$ if $i < i'$ and $j < j'$. We refer to elements of $\mathbf{IND}_x \times \mathbf{IND}_y$ as *index pairs* or simply *pairs*.

There are different variations on the notion of edit distance of two strings. In the most common version, the edit distance of x, y is the minimum number of insertions, deletion, or substitutions needed to transform one of the strings to the other. (This quantity is symmetric.) For our purposes we deal with a variant where only deletions are allowed, and symbols may be deleted from either string, with the goal of making the two strings the same. Throughout this paper $\mathbf{edit}(x,y)$ refers to this variant. It is well known, and easy to show that this variant is bounded between the usual edit distance and twice the usual edit distance. Since we are interested in weak approximations (whose approximation factor is a large constant or super-constant) the factor 2 is not important and all our results apply to the more common variant up to this factor of 2.

An index pair (i,j) is a *matched pair* if $x[i] = y[j]$. An *ordered matching* is a set M of matched pairs that can be ordered as $(i_1, j_1), \ldots, (i_k, j_k)$ such that $(i_1, j_1) \ll \cdots \ll (i_k, j_k)$.

An x-index (resp. y-index) is *unmatched* in M if it is not the first (resp. second) coordinate of any pair of M. Note that by deleting the unmatched symbols we make the two strings the same. The *edit distance* (with respect to deletions in both strings), denoted $\mathbf{edit}(x,y)$ is the minimum number of unmatched x-indices and y-indices in any ordered matching.

4 Single pass algorithms for ordered matchings

We introduce a class of simple algorithms, called *single pass algorithms*, for finding an ordered matching of two input strings x, y. For such algorithms, we imagine the two strings being presented to us from left to right one symbol at a time, and we build an ordered matching one pair at a time. The state of the algorithm is given by a pair $\mathbf{current} = (\mathbf{current}_x, \mathbf{current}_y)$ of an x-index and a y-index. Initially $\mathbf{current} = (1,1)$. The algorithm proceeds through a sequence of steps. In each step it processes the pair $\mathbf{current}$. If $x[\mathbf{current}_x] = y[\mathbf{current}_y]$ then $\mathbf{current}$ is added to the matching and both $\mathbf{current}_x$ and $\mathbf{current}_y$ are increased by one. (This is called a *match step*.) Otherwise the step is a *mismatch step*, and the algorithm must make a choice s which is either x or y. A mismatch step is said to be an *x-step* or a *y-step* depending on the string selected. Choosing string s means that the algorithm decides to leave $\mathbf{current}_s$ of string s unmatched and increases the index $\mathbf{current}_s$ by 1.

The algorithm keeps track of the total number of mismatch steps (and possibly other information). Prior to each step, the algorithm has committed to an ordered matching M from the set $\{1, \ldots, \mathbf{current}_x - 1\} \times \{1, \ldots, \mathbf{current}_y - 1\}$ and also committed to leaving any x-index in $\{1, \ldots, \mathbf{current}_x - 1\} \setminus M_x$ unmatched and any y-index in $\{1, \ldots, \mathbf{current}_y - 1\} \setminus M_y$ unmatched. Notice that the number of indices declared unmatched is equal to the number of mismatches prior

to the current step.

The algorithm terminates when $\mathbf{current}_x$ reaches $1 + |x|$ or $\mathbf{current}_y$ reaches $1 + |y|$. Upon termination, we say the indices in $\{\mathbf{current}_x, \ldots, |x|_x\}$ and $\{\mathbf{current}_y, \ldots, |y|_x\}$ are *unprocessed*. (One of these sets is empty.) All unprocessed indices are unmatched and there are $(|y| + 1 - |\mathbf{current}_y|) + (|x| + 1 - \mathbf{current}_x)$ of them. The output of the algorithm is the number of mismatch steps plus the number of unprocessed indices.

To fully specify the algorithm requires a *selection rule* which determines the choice of x or y for each mismatch step. In general, the selection rule may depend on the entire prior history of the algorithm. We are especially interested in selection rules called *basic selection rules* whose choice depends only on the number of mismatches seen so far. A basic selection rule is thus specified by a string σ with entries from $\{\mathtt{x}, \mathtt{y}\}$, where the choice for the u-th mismatch is given by $\sigma[u]$. Here, $\mathtt{x}$ and $\mathtt{y}$ are symbols that refer to the strings x and y, respectively. The deterministic single pass algorithm with selection rule σ is denoted A_σ. Algorithm 1 provides pseudo-code for a deterministc single pass algorithm with the selection rule γ. Except for the specific choice of the selection rule, the algorithm is the same for any single pass algorithm. The selection rule γ is the one that will be used to establish Theorem 5.1.

To carry out the above procedure σ has to be long enough, depending on x and y. It suffices that σ contains at least $|x|$ occurrences of $\mathtt{x}$ and at least $|y|$ occurrences of $\mathtt{y}$.

A randomized single pass algorithm is given by a *randomized selection rule*, which is a probability distribution $\widetilde{\sigma}$ over selection rules.

We denote the output of the algorithm with selection rule σ on inputs x, y by $\mathbf{edit}_\sigma(x, y)$. From the definition of the algorithm, the output u counts the number of indices of x and y that are left unmatched by M, and so $\mathbf{edit}_\sigma(x, y)$ is an upper bound on $\mathbf{edit}(x, y)$. It is not hard to show that for every pair x, y there is a basic selection rule σ, depending on x, y, such that $\mathbf{edit}_\sigma(x, y) = \mathbf{edit}(x, y)$.

We are interested in determining how well a fixed basic selection rule (possibly randomized) can perform on all inputs x, y. Saha showed that the randomized selection rule $\widetilde{\rho}$ in which each successive mismatch step is selected independently to be $\mathtt{x}$ or $\mathtt{y}$ with probability $1/2$ has the property that with high probability $\mathbf{edit}_{\widetilde{\rho}}(x, y) = O(\mathbf{edit}(x, y)^2)$. In the next section we will give a deterministic selection rule γ that achieves this same bound. On the other hand in Section 6 we show this is best possible: for any (possibly randomized) selection rule $\widetilde{\sigma}$ and any d there exists an input x, y with $\mathbf{edit}(x, y) \leq d$ such that with probability close to 1, $\mathbf{edit}_{\widetilde{\sigma}}(x, y) = \Omega(d^2)$.

Let σ be a deterministic selection rule, and x and y be input strings. We introduce various quantities that track the progress of the algorithm A_σ applied to x and y. These definitions will be used for both our upper and lower bounds.

- $c(h) = (c_x(h), c_y(h))$ denotes the value of **current** at the beginning of step h of the algorithm.
- The *offset* at step h is the quantity $\delta(h) = c_x(h) - c_y(h)$. We set $\delta(0) = 0$.
- Let $u(h)$ be the number of mismatch steps through the end of step h.
- Let $u_x(h)$ (resp. $u_y(h)$) be the number of x-steps (resp. y-steps) through the end of step h. Set $u_x(0) = u_y(0) = 0$. Obviously $u(h) = u_x(h) + u_y(h)$.
- Let $\Delta_\sigma(w)$ be the number of $\mathtt{x}$ entries minus the number of $\mathtt{y}$ entries in positions $\{1, \ldots, w\}$ of σ. Set $\Delta_\sigma(0)$ to 0.

Proposition 4.1. *For any step h we have $\delta(h) = u_x(h-1) - u_y(h-1) = \Delta_\sigma(u(h-1))$.*

ALGORITHM 1: $\mathbf{edit}_\gamma(x, y)$

Input: Strings x, y.
Output: $\mathbf{edit}_\gamma(x, y)$.

Set $\gamma = \mathtt{xyyyxxxxxyyyyyyy} \cdots \mathtt{x}^{4i-3}\mathtt{y}^{4i-1} \cdots$;
Initialization: $\mathbf{current}_x = 1$, $\mathbf{current}_y = 1$, $u = 0$;
while $\mathbf{current}_x < 1 + |x|$ *and* $\mathbf{current}_y < 1 + |y|$ **do**
  if $x[\mathbf{current}_x] = y[\mathbf{current}_y]$ **then**
    $\mathbf{current}_x = \mathbf{current}_x + 1$ and $\mathbf{current}_y = \mathbf{current}_y + 1$;
  end
  else
    $u = u + 1$;
    if $\gamma[u] = \mathtt{x}$ **then** $\mathbf{current}_x = \mathbf{current}_x + 1$;
    else $\mathbf{current}_y = \mathbf{current}_y + 1$;
  end
end
Output $u + |x| - \mathbf{current}_x + |y| - \mathbf{current}_y + 2$.

Proof. Letting $M(h)$ denote the matching selected during the first h steps, we have $c_x(h) = |M(h-1)| + u_x(h-1)$ and $c_y(h) = |M(h-1)| + u_y(h-1)$, so the first equality follows. For the second equality we note that $u_x(h-1)$ (resp. $u_y(h-1)$) is equal to the number of $\mathtt{x}$'s (resp. $\mathtt{y}$'s) up to position $u(h-1)$ in σ. □

5 A deterministic basic selection rule giving a quadratic approximation

As mentioned earlier, Saha showed that a uniformly random selection rule $\widetilde{\rho}$ on input x, y with $\mathbf{edit}(x, y) = d$ has $\mathbf{edit}_{\widetilde{\rho}}(x, y) = O(d^2)$ with high probability. In this section we attain the same performance with a basic deterministic selection rule. Define the selection rule γ to consist of alternating blocks of $\mathtt{x}$'s and $\mathtt{y}$'s, where the j-th block has length $2j - 1$ and is an x-block if j is odd and a y-block if j is even. (See Algorithm 1.) Define *block* j of γ to be the set of indices corresponding to the j-th block.

The main result of this section is:

Theorem 5.1. *On any input x, y, $\mathbf{edit}_\gamma(x, y)$ is at most $\mathbf{edit}(x, y)^2 + 4 \cdot \mathbf{edit}(x, y)$.*

We provide some intuition behind our algorithm, making some simplifying assumptions to avoid technical complications.

Our goal is to show that A_γ outputs an estimate $O(d^2)$ on inputs x and y of edit distance d. Let M^* be an optimal matching between x and y. The algorithm doesn't know M^* but the goal is to have the algorithm follow M^* as much as possible. We think of d as "small" since if d is larger than $\sqrt{n}$ we don't care what A_γ does. Since d is small, the picture of M^* is that it consists mostly of long perfect runs of consecutive matched pairs. Within each run, all of the matched pairs have the same *offset*, where the offset is the x-index minus the y-index.

If the algorithm manages to synchronize with M^* (so that $(\mathbf{current}_x, \mathbf{current}_y) \in M^*$) then it will stay synchronized as long as the current perfect run continues. When the current perfect run

ends, the algorithm and M^* may become unsynchronized, but M^* will have at least one unmatched index at the end of that run. Our goal is to show that the algorithm doesn't encounter too many mismatches before becoming synchronized again. What we are aiming for (roughly) is that the j-th time that the algorithm becomes unsynchronized from M^*, it will require at most $O(j)$ mismatch steps to resynchronize. Thus, after becoming unsynchronized for the j-th time, the algorithm has had $O(j^2)$ mismatch steps while M^* has at least j unmatched indices.

The blocks of γ are designed to attain this synchronization. The j-th block consists of $2j-1$ `x`'s or `y`'s. We refer to the portion of the algorithm where the j-th block of γ is being used as *phase* j. The goal (roughly) is that for each phase, the algorithm should become synchronized with M^* sometime during the phase, or if not, it is because M^* already has a total of at least j unmatched indices among the indices visited by the algorithm through the end of phase j. Thus after phase j, the algorithm has had j^2 mismatch steps, and M^* will have had at least j mismatches.

As mentioned, in the "typical" situation we expect M^* to consist of relatively few long perfect runs of matched edges all having the same offset. Consider a perfect run that includes the x- or y-index at the beginning of phase j and let i be the common offset of the pairs in the run. If $|i| \geq j$ then there must already be at least j unmatched indices of M^* before the perfect run starts. So assume $i \in [1-j, j-1]$. The definition of γ ensures that for odd j the offset $\mathbf{current}_x - \mathbf{current}_y$ during phase j climbs from $1-j$ to $j-1$, and for even j, the offset falls from $j-1$ to $1-j$. In either case, at some point during the phase the offset will be i and the algorithm will be synchronized with M^*.

The above sketch is based on the picture of M^* as consisting of a small number of long "perfect runs", which is an oversimplification. The actual argument requires some careful considerations and is presented in Section 8.

5.1 Some remarks on derandomizing applications of the random single pass algorithm.

The deterministic algorithm presented above has similar approximation guarantees for edit distance as Saha's random single pass algorithm. In the introduction, it was mentioned that Saha's algorithm appears as an ingredient in some other applications related to edit distance. One might hope that by replacing Saha's algorithm by the above deterministic algorithm these applications could be derandomized, but this does not seem to be the case.

For example, Kociumaka and Saha [26] gave a sublinear time approximation algorithm which for a parameter $p \in [1, d]$ gives an $O(dp)$-factor approximation to edit distance in time $O(n/p)$. Their algorithm generalizes Saha's algorithm as follows: For a parameter $p < d$, their algorithm proceeds similarly to our simple one pass algorithms: it moves pointers in strings x and y from left to right but instead of moving them by one step at a time, at each time step it picks a random jump of expected length $O(p)$ and moves them by this amount. Additionally if there is a mismatch then it selects one of the pointers at random and advances the pointer by one. Hence, the algorithm runs in expected time $O(n/p)$ and provides $O(dp)$-approximation to edit distance. The algorithm makes random choices in two places: to select the length of the jump and to select which pointer to advance on a mismatch. One could use our selection rule (the string γ) to chose which pointer to advance on a mismatch to partially derandomize the algorithm. However, the other randomness is essential as there cannot be any deterministic sublinear time algorithm for edit distance (not even for testing string identity).

One application of the random single pass algorithm that was mentioned in the introduction

was the randomized reduction of Dyck edit distance to standard edit distance. It turns out that this reduction can be derandomized using our deterministic single pass algorithm. But in working out the details we found a significant simplification of the reduction that eliminates the need for either the randomized or deterministic single pass algorithm. This will be presented in Section 7.

6 Limits on the approximation quality of basic selection rules

In Section 5 we gave a basic selection rule that gives a quadratic approximation to edit distance. Earlier, Saha showed that the uniform random selection rule gives a quadratic approximation with high probability. In this section we show that no (possibly randomized) basic selection rule can do better than a quadratic approximation.

Theorem 6.1. *For any (possibly randomized) selection rule $\widetilde{\sigma}$, for any $d \geq 1$, for any $\epsilon > 0$ there is a pair of strings x, y with $\mathbf{edit}(x, y) \leq 6d$ such that:*

$$\Pr[\mathbf{edit}_{\widetilde{\sigma}}(x, y) \leq \epsilon d^2] < \epsilon.$$

The proof will follow some preliminaries. Let d be given and define:

- $n = 2d^2$
- Σ be the alphabet consisting of $n + 2d + 2$ letters, $\{\mathsf{c}_{-d}, \mathsf{c}_{-d+1}, \ldots, \mathsf{c}_0, \ldots, \mathsf{c}_{n+d}, \mathsf{c}\}$.
- x is the string obtained from the string $\mathsf{c}_1, \ldots, \mathsf{c}_n$ by replacing all of the symbols c_{jd} by c for $j \in \{1, \ldots, 2d\}$.
- For each $i \in [-d, d]$, y_i is the string $\mathsf{c}_{1+i}\mathsf{c}_{2+i} \cdots \mathsf{c}_{n+i}$.

Clearly $\mathbf{edit}(x, y_i) \in [4d, 6d]$ for each i. We will prove:

Lemma 6.2. *Let σ be a deterministic selection rule. For $q \leq d$, let $I_q = \{i \in [-d, d] : \mathbf{edit}_\sigma(x, y_i) < qd\}$. Then $|I_q| \leq q + 1$.*

The proof will make use of notation and Proposition 4.1 from Section 4.

For each integer $i \in [-d, \ldots, d]$ define $L_i = \{w \geq 0 : \Delta_\sigma(w) = i\}$. Clearly the sets L_i are pairwise disjoint. For $i \in [-d, d]$ let $N_i = L_i \cap \{0, \ldots, \lceil qd \rceil - 1\}$.

Lemma 6.3. *For $i \in I_q$, we have $|N_i| \geq d$.*

Proof. Fix $i \in I_q$ and consider the execution of A_σ on (x, y_i). Divide the string x into blocks $x^1, x^2, \ldots, x^{2d}$ where each block has length d. Each block ends with the symbol c.

Let S be the set of steps of $A_\sigma(x, y_i)$ that find a match. Since all matching pairs of indices for x and y_i are of the form $(j + i, j)$ for some j, for any $h \in S$ we have $c_x(h) = c_y(h) + i$ thus $\delta(h) = i$. By Proposition 4.1 we have $\Delta_\sigma(u(h-1)) = i$, and thus $u(h-1) \in N_i$, where $u(h-1)$ is the number of mismatches found through the end of step $h - 1$.

Call a block of x *good* if the run of $A_\sigma(x, y)$ matches at least one symbol in the block and *bad* otherwise. Since $\mathbf{edit}_\sigma(x, y_i) < qd \leq d^2$ there are at least d good blocks $j_1 < \cdots < j_d$. For $1 \leq s \leq d$ let h_s be the first step of the algorithm that finds a match in block j_s. By the previous paragraph we have $u(h_s - 1) \in N_i$ for each s. But we also have $u(h_1 - 1) < u(h_2 - 1) < \cdots u(h_d - 1)$ since each block ends with a c which adds a mismatch. Therefore $|N_i| \geq d$, as required. □

Proof of Lemma 6.2. From Lemma 6.3 we have

$$|I_q|d \leq \sum_{i \in I_q} |N_i| \leq \lceil qd \rceil \leq (q+1)d,$$

where the second inequality comes from the fact that N_i are disjoint subsets of $\{0, \ldots, \lceil qd \rceil - 1\}$. Therefore $|I_q| \leq q+1$. □

Proof of Theorem 6.1. Fix $\epsilon > 0$. If $\epsilon < 4/d$ then $\Pr[\mathbf{edit}_{\widetilde{\sigma}}(x, y_i) < \epsilon d^2] = 0$ since $\epsilon d^2 < 4d \leq \mathbf{edit}(x, y_i)$ and the output of any single pass algorithm is at least the edit distance. So assume $\epsilon \geq 4/d$ and let $q = \epsilon d$. Let Z_i be the random variable that is 1 if $\mathbf{edit}_{\widetilde{\sigma}}(x, y_i) < \epsilon d^2$ and 0 otherwise, and set $p_i = \Pr[\mathbf{edit}_{\widetilde{\sigma}}(x, y_i) < \epsilon d^2] = \mathbb{E}_{\widetilde{\sigma}}[Z_i]$. Let $Z = \sum_{i=-d}^{d} Z_i$, and note that Lemma 6.2 implies $Z \leq q+1$ and so $q+1 \geq \mathbb{E}_{\widetilde{\sigma}}[Z] = \sum_{i=-d}^{d} p_i$. Therefore there is an i such that $p_i \leq \frac{q+1}{2d+1} = \frac{\epsilon d+1}{2d+1}$ and since $\epsilon d \geq 4$ this is at most $\frac{2\epsilon d}{2d+1} < \epsilon$. □

We believe that it should be possible to make our lower bound work for the case of strings over binary alphabets, possibly with a loss of log factors. Indeed, if x is a randomly chosen string then any of its shifts by i symbols will have edit distance $\Omega(i)$. Flipping d well-spaced bits in each such shift will give strings y_i with properties similar to our strings x and y_i. This would seem to require non-trivial technical work to work out the details.

7 A simple and deterministic reduction from Dyck edit distance approximation to edit distance approximation

Given any language L, the edit distance problem for L (with respect to insertions and deletions) is: given a string x, what is the minimum number of symbol insertions and deletions that need to be applied to x to put it in L. A well-studied case of this problem is the case where L is a Dyck language; this is known as the *Dyck edit distance problem*.

The Dyck language is defined as follows. Let T be a finite set and let $L(T)$ be the set of symbols of the form $(_t$ for each $t \in T$ (*left parentheses*) and $R(T)$ be the set of symbols of the form $)_t$ for $t \in T$ (*right parentheses*). Let $\hat{T}$ be the set $L(T) \cup R(T)$; as usual, $\hat{T}^*$ is the set of finite strings over $\hat{T}$. For any $t \in T$, the ordered pair $(_t,)_t$ is called a matched pair of parentheses. If x is a string in $\hat{T}^*$, a *removable pair* is a pair of consecutive entries that form a matched pair. The Dyck language $D(T)$ for T is the set of strings in $\hat{T}^*$ that can be reduced to the empty string by successively removing removable pairs. The Dyck edit distance $\mathbf{edit}_D(x)$ is the minimum number of symbol additions or deletions that transform x to a string in $D(T)$. It is not hard to show that if $\mathbf{edit}_D(x) = k$ then x can be transformed into a string in $D(T)$ by k deletions, that is, insertions are not needed.

The following equivalent formulation of Dyck edit distance will be more convenient. Given a string $x \in \hat{T}^*$, a *non-crossing matching of* x is a set of ordered pairs $\{(i_1, j_1), \ldots, (i_k, j_k)\}$ of distinct indices from $\{1, \ldots, |x|\}$ such that (1) $i_h < j_h$ for each h, (2) $x[i_h], x[j_h]$ is a matched pair of parentheses for each h, (3) There are no *crossing pairs*, where $(i_h, j_h), (i_r, j_r)$ are said to cross if $i_h \leq i_r \leq j_h \leq j_r$ or $i_r \leq i_h \leq j_r \leq j_h$. The Dyck language is the set of strings that admit a non-crossing perfect matching. The Dyck edit distance $\mathbf{edit}_D(x)$ is the minimum, over all non-crossing matchings of x, of the number of unmatched indices. It is well known, and easy to verify, that the two definitions of $\mathbf{edit}_D$ are the same.

Say that a string of parentheses is an *LR-string* if no left parenthesis follows any right parenthesis. An LR-string consists of a (possibly empty) string of left parentheses followed by a (possibly empty) string of right parentheses. It is well known that the ordinary edit distance problem for a pair of strings x, y is equivalent to the Dyck edit problem for LR strings under the following bijective correspondence: map a pair x, y in T^* to the string $s(x,y) \in \hat{T}^*$ given by $s(x,y) = L(x)R(\overleftarrow{y})$ where $L(x)$ is obtained by replacing each entry of x by the corresponding left parenthesis, $\overleftarrow{y}$ is y in reverse order, and $R(\overleftarrow{y})$ is obtained by replacing each entry of $\overleftarrow{y}$ by the corresponding right parenthesis. Furthermore, this reduction preserves approximations so that an approximation algorithm for standard edit distance gives a similar approximation for Dyck edit distance for LR strings, and vice versa.

Saha [31] gave a randomized reduction from the approximation of Dyck edit distance for a general string $x \in \hat{T}$ to the approximation of Dyck edit distance for LR-strings (or equivalently to the approximation of standard (two string) edit distance). The reduction and its analysis are rather involved. Here we present a much simplified variation on Saha's reduction. The property of the reduction is stated in Theorem 7.1 and Corollary 7.2. The asymptotic running time and the approximation factor are similar to (and slightly better than) that obtained in [31], and the reduction is deterministic rather than randomized. The main advantage is that both the algorithm and analysis are extremely simple.

We need some additional definitions. A *maximal LR-segment* of a string x is an LR-string that is a substring of x and can not be extended to a larger LR-substring. It is easy to see that the maximal LR-segments give a partition of x into disjoint substrings, called the *LR-decomposition of* x. Say that an LR-string is *two-sided* if it contains at least one left parenthesis and one right parenthesis and is *one-sided* otherwise. Every segment in the LR-decomposition of x is two-sided except possibly the first (which might have no left parentheses) and the last (which might have no right parentheses).

We define $h(x)$, the *LR-height* of x, to be the number of two-sided segments in the LR-decomposition of x. Thus height $h(x) \leq |x|/2$.

Theorem 7.1. *There is a deterministic algorithm that on input a string $x \in \hat{T}^*$ runs in $\tilde{O}(|x|)$ time and outputs a collection $\mathcal{C}(x)$ of LR-strings satisfying $\sum_{z \in \mathcal{C}(x)} |z| = |x|$ such that:*

P1. $\mathbf{edit}_D(x) \leq \sum_{z \in \mathcal{C}(x)} \mathbf{edit}_D(z)$

P2.

$$\sum_{z \in \mathcal{C}(x)} \mathbf{edit}_D(z) \leq \left\{ \begin{array}{ll} \mathbf{edit}_D(x), & \text{if } h(x) = 0 \\ (3 + 2\log h(x)) \cdot \mathbf{edit}_D(x) & \text{if } h(x) \geq 1 \end{array} \right\}$$

(Here and elsewhere in the section, all logarithms are base 2.)

This algorithm provides a reduction of the estimation of $\mathbf{edit}_D(x)$ to the estimation of $\mathbf{edit}_D(z)$ for all $z \in \mathcal{C}(x)$. Since each such z is an LR-string, $\mathbf{edit}_D(z)$ is equivalent to an ordinary edit distance computation.

Saha [31] observed that the quantity $h(x)$ in the above bound can be replaced by $\mathbf{edit}_D(x)$. Let $\bar{x}$ be the string obtained from x by successively removing removable pairs until none remain. It is easy to show that $\mathbf{edit}_D(\bar{x}) = \mathbf{edit}_D(x)$. Also, $\mathbf{edit}_D(\bar{x}) \geq h(\bar{x})$ since for each LR-block in the LR-decomposition, either the final left parenthesis or the first right parenthesis of the block must be unmatched in any non-crossing matching. By preprocessing x to reduce it to $\bar{x}$ (which can be

done easily in linear time), and applying the algorithm to $\bar{x}$, we can replace the $1+2\log h(x)$ factor by $1+2\log \mathbf{edit}_D(x)$.

Suppose A is any algorithm that on input two strings provides an upper bound on the standard edit distance between the two strings. By the equivalence between the Dyck edit distance for LR strings and standard edit distance of two strings mentioned earlier, we can view A as an algorithm that takes as input an LR-string of parentheses and outputs an upper bound on the Dyck edit distance of the string. Define the algorithm B_A for strings x of arbitrary height to be the algorithm that first removes from x all matching pairs of consecutive parenthesis to get $\bar{x}$, then constructs $\mathcal{C}(\bar{x})$ and outputs $\sum_{z\in\mathcal{C}(\bar{x})} A(z)$. As an immediate consequence of Theorem 7.1 we have:

Corollary 7.2. *Let A be an algorithm for LR-strings as above. Suppose $\beta(n,d)$ and $t(n,d)$ are functions such that for any LR-string z with $|z|\le n$ and $\mathbf{edit}_D(z)\le d$, $A(z)$ runs in time at most $|z|\cdot t(n,d)$ and $\mathbf{edit}_D(z)\le A(z)\le \beta(n,d)\cdot \mathbf{edit}_D(z)$. Then for any string $x\in\hat{T}^*$ with $|x|\le n$ and $\mathbf{edit}_D(x)\le d$, $B_A(x)$ runs in time $\tilde{O}(n)+n\cdot t(n,d)$ and satisfies:*

$$\mathbf{edit}_D(x)\le B_A(x)\le \beta(n,d)\cdot(3+2\log \mathbf{edit}_D(x))\cdot \mathbf{edit}_D(x).$$

We now describe the algorithm for Theorem 7.1 that finds the collection $\mathcal{C}(x)$. The algorithm works recursively. If x is the null string then $\mathcal{C}(x)$ is empty. For x non-null:

1. Construct the LR-decomposition $\mathcal{D}$ of x.
2. Construct the collection $\mathcal{E}$ of LR-strings as follows:
   (a) If the first or last segment of $\mathcal{D}$ is one-sided then put that segment in $\mathcal{E}$.
   (b) For each two-sided segment $y\in\mathcal{D}$: let $m(y)$ be the minimum of the number of left and right parentheses in y and let $z(y)$ be the (unique) consecutive substring of y consisting of $m(y)$ left parentheses and $m(y)$ right parentheses. Add $z(y)$ to $\mathcal{E}$.
3. Let x' be the string obtained from x by deleting all of the segments of $\mathcal{E}$, and recursively construct $\mathcal{C}(x')$.
4. Set $\mathcal{C}(x)=\mathcal{E}\cup\mathcal{C}(x')$.

A simple induction on $|x|$ shows that $\sum_{z\in\mathcal{C}(x)}|z|=|x|$: From the algorithm description $|x|=|x'|+\sum_{z\in\mathcal{E}}|z|$. By induction this is equal to $\sum_{z\in\mathcal{C}(x')}|z|+\sum_{z\in\mathcal{E}}|z|=\sum_{z\in\mathcal{C}(x)}|z|$.

We now establish that $\mathcal{C}(x)$ satisfies P1 and P2.

P1 is also proved by a simple induction on $|x|$. The basis $|x|=0$ is trivial. Suppose $|x|\ge 1$ and let x' and $\mathcal{E}$ be as defined in the algorithm. Select optimal non-crossing matchings for x' and for each $z\in\mathcal{E}$. The union of these matchings is a noncrossing matching for x and therefore $\mathbf{edit}_D(x)\le \mathbf{edit}_D(x')+\sum_{z\in\mathcal{E}}\mathbf{edit}_D(z)$. By induction $\mathbf{edit}_D(x')\le\sum_{z\in\mathcal{C}(x')}\mathbf{edit}_D(z)$ and so $\mathbf{edit}_D(x)\le\sum_{z\in\mathcal{C}(x)}\mathbf{edit}_D(z)$ as required to establish P1.

P2 requires more work. It is proved by induction on $h(x)$. If $h(x)=0$ then $x=wy$ where w is a string or right parentheses and y is a string of left parentheses. The set $\mathcal{C}(x)$ consists of the strings w and y. In this case we have $\mathbf{edit}_D(x)=|x|$, $\mathbf{edit}_D(y)=|y|$ and $\mathbf{edit}_D(w)=|w|$, so $\sum_{z\in\mathcal{C}(x)}\mathbf{edit}_D(z)=|y|+|w|=\mathbf{edit}_D(x)$, to establish the base case.

So assume $h(x)\ge 1$. We first note that $h(x')\le h(x)/2$. Let $y^1,\ldots,y^{h(x)}$ be the sequence of two-sided segments of the LR-decomposition of x. Let w^i be the segment of y^i obtained by

deleting $z(y^i)$ (where $z(y^i)$ is as defined in Step 2b of the algorithm). The string x' is equal to the concatenation of $w^1, \ldots, w^{h(x)}$. Since each non-null w^i is one-sided, every two-sided segment in the decomposition of x' is a concatenation of 2 or more w^i segments and therefore $h(x') \leq h(x)/2$.

Applying the induction hypothesis to x' yields:

$$\sum_{z \in \mathcal{C}(x')} \mathbf{edit}_D(z) \leq \left\{ \begin{array}{ll} \mathbf{edit}_D(x'), & \text{if } h(x') = 0 \\ (3 + 2\log h(x')) \cdot \mathbf{edit}_D(x') & \text{if } h(x') \geq 1 \end{array} \right\}$$

We claim that the two expressions on the right hand side are bounded above by $(1 + 2\log h(x)) \cdot \mathbf{edit}_D(x')$. If $h(x') = 0$ then $\mathbf{edit}_D(x') \leq (2\log h(x) + 1) \cdot \mathbf{edit}_D(x')$ since $\log h(x) \geq 0$. If $h(x') \geq 1$ then since $h(x) \geq 2h(x')$ we have $3 + 2\log h(x') \leq 1 + 2\log h(x)$

To complete the induction we will prove:

Lemma 7.3.

1. $\mathbf{edit}_D(x') \leq \mathbf{edit}_D(x)$
2. $\sum_{z \in \mathcal{E}} \mathbf{edit}_D(z) \leq 2 \cdot \mathbf{edit}_D(x)$

The lemma suffices to complete the induction step for P2:

$$\begin{aligned} \sum_{z \in \mathcal{C}(x)} \mathbf{edit}(z) &= \sum_{z \in \mathcal{C}(x')} \mathbf{edit}_D(z) + \sum_{z \in \mathcal{E}} \mathbf{edit}_D(z) \\ &\leq (2\log h(x) + 1) \cdot \mathbf{edit}_D(x) + 2 \cdot \mathbf{edit}_D(x) = (2\log h(x) + 3) \cdot \mathbf{edit}_D(x), \end{aligned}$$

as required.

So it suffices to prove Lemma 7.3.

Proof. Let M be a maximum non-crossing matching for x. For $z \in \mathcal{E} \cup \{x'\}$, let M_z be the set of pairs of M with both ends in z and let N_z be the set of pairs with one end in z and one end out, and let U_z be the set of indices corresponding to z that are unmatched in M. Note that $\mathbf{edit}_D(x) \geq |U_{x'}| + \sum_{z \in \mathcal{E}} |U_z|$.

Proposition 7.4. *For each $z \in \mathcal{E}$:*

1. $\mathbf{edit}_D(z) \leq |U_z| + |N_z|$
2. *The indices of z that belong to pairs of N_z are either all left parentheses or all right parentheses.*
3. $|U_z| \geq |N_z|$.

Proof. For the first part, we note that the number of indices of z that are left unmatched by M_z is $|U_z| + |N_z|$.

For the second part, any pair in N_z that includes an index to a left parenthesis of z must have its other end after z, and any pair in N_z that includes an index to a right parenthesis of z must have its other end before z. If there were a pair of each type then, since z is an LR string, they would cross, which is a contradiction that proves the second part.

For the third part, we note that this is trivial for a one-sided block since $N_z = \emptyset$. So consider a two-sided block. By the second part of the proposition, we assume, without loss of generality, that

all indices of z that are matched in N_z correspond to left parentheses, and note that z has $|z|/2$ left indices and $|z|/2$ right indices. We then have:

$$|N_z| + |M_z| \leq |z|/2 \leq |U_z| + |M_z|.$$

where the first inequality comes from the fact that the pairs in $N_z \cup M_z$ correspond to distinct left parentheses of z and the second inequality comes from the fact that every right parenthesis of z either belongs to U_z or is paired in M_z. The third part of the proposition follows. □

Continuing with the proof of the lemma: For the first part, we note that $\mathbf{edit}_D(z)$ is at most the number of indices of x' that are unmatched in $M_{x'}$. The indices that are not matched in N'_x are the indices in $U_{x'}$ (those indices of x' unmatched in M) together with the at most $\sum_{z\in\mathcal{E}} |N(z)|$ indices of x' that belong to pairs from $\bigcup_z N_z$.

Applying part 3 of the proposition we then establish the first part of the lemma:

$$\mathbf{edit}_D(x') \leq |U_{x'}| + \sum_{z\in\mathcal{E}} |N(z)| \leq |U_{x'}| + \sum_{z\in\mathcal{E}} |U(z)| \leq \mathbf{edit}_D(x).$$

To prove the second part of the lemma, we apply parts 1 and 3 of the proposition to get:

$$\sum_{z\in\mathcal{E}} \mathbf{edit}_D(z) \leq \sum_{z\in\mathcal{E}} |U_z| + |N_z| \leq \sum_{z\in\mathcal{E}} 2|U_z| \leq 2 \cdot \mathbf{edit}_D(x).$$

□

As observed earlier, Lemma 7.3 completes the proof of Theorem 7.1.

8 Proof of Theorem 5.1

8.1 Preliminaries

To prove the main theorem we need to observe some simple facts. We use notation from Section 4. Recall, our selection rule γ consists of alternating blocks of `x`'s and `y`'s, where the j-th block has length $2j-1$ and is an x-block if j is odd and a y-block if j is even. We note some elementary properties of γ. (The rudimentary induction proofs are omitted.)

Proposition 8.1.

1. *Block j of γ starts at position $(j-1)^2+1$ and ends in position j^2.*
2. *For $0 \leq q < 2j-1$, $\Delta_\gamma((j-1)^2+q) = (-1)^j \cdot (j-1-q)$. In particular, for any $t < j^2$, $|\Delta_\gamma(t)| < j$.*

The following two propositions give simple criteria for showing the presence of unmatched indices in an ordered matching:

Proposition 8.2. *Let M be a matching. Suppose $i < j$.*

1. *If M contains a matched pair (i', j') with $i' \leq i$ and $j' \geq j$ then there are at least $j - i$ unmatched y-indices in $\{1, \ldots, j-1\}$.*

2. *If M contains a matched pair (j', i') with $i' \leq i$ and $j' \geq j$ then there are at least $j - i$ unmatched x-indices in $\{1, \ldots, j-1\}$.*

Proof. For the first part, since j'_y is matched to i'_x and $j \leq j'$, y-indices in $\{1, \ldots, j-1\}$ can only be matched to x-indices in $\{1, \ldots, i'-1\}$ so at least $j - i' \geq j - i$ y-indices in $\{1, \ldots, j-1\}$ are unmatched. The second part is proved similarly. □

Proposition 8.3. *Let M be a matching. Let $i < i'$ and suppose (i, j) and (i', j') are index pairs having the same offset, i.e. $i - j = i' - j'$. If $(i, j) \in M$, and $(i', j') \notin M$ then some x-index in $\{i+1, \ldots, i'\}$ or y-index in $\{j+1, \ldots, j'\}$ is unmatched by M.*

Proof. Let $c = i - j$. Let ℓ be the least index in $[j, j']$ such that $(\ell + c, \ell) \notin M$. Then $(\ell + c - 1, \ell - 1) \in M$ and so $(\ell + c)_x$ can only be matched to a y-index $> \ell$ (otherwise the pair would cross $(\ell + c - 1, \ell - 1)$) and similarly ℓ_y can only be matched to an x-index $> \ell + c$. Thus if both are matched in M, then their pairs would cross, so one of them is unmatched. □

8.2 Proof of Theorem 5.1

We are ready to prove the main theorem of this section. Consider the execution of A_γ on x, y and let M be the resulting ordered matching. Let $k = k(x, y)$ be the number of completed blocks of γ when we run $A_\gamma(x, y)$. Thus k is the largest integer j so that $A_\gamma(x, y)$ has at least j^2 mismatch steps. Following notation from Section 4, $c(h) = (c_x(h), c_y(h))$ is the value of **current** at the beginning of step h and $\delta(h) = c_x(h) - c_y(h)$.

We first prove an upper bound on the cost of $A_\gamma(x, y)$.

Lemma 8.4. $\mathbf{edit}_\gamma(x, y) \leq k^2 + 3k + |(|x| - |y|)|$.

Proof. Let h be the step number at termination and let u be the number of mismatches at termination. $\mathbf{edit}_\gamma(x, y)$ is equal to u plus the number of indices of x or y that are unprocessed at termination. Since the algorithm terminates before γ reaches the end of block $k+1$, we have $u \leq (k+1)^2 - 1 = k^2 + 2k$. So it suffices to show that the number of unprocessed indices is at most $|(|x| - |y|)| + k$.

The algorithm terminates with $c_x(h) = 1 + |x|$ or $c_y(h) = 1 + |y|$. Assume, without loss of generality, that $\mathbf{current}_x = |x| + 1$. The number of unprocessed indices is $|y| + 1 - c_y(h) = (|y| - |x|) + c_x(h) - c_y(h) = |y| - |x| + \delta(h)$. By Proposition 4.1, $\delta(h) = \Delta_\gamma(u)$ which, by Proposition 8.1 is at most k since $u < (k+1)^2$. □

Theorem 5.1 will follow if $\mathbf{edit}(x, y) \geq \max(k, |(|x| - |y|)|)$. Trivially, $\mathbf{edit}(x, y) \geq |(|x| - |y|)|$, so it remains to show that $\mathbf{edit}(x, y) \geq k$. Let M^* be an optimal ordered matching. We will show that M^* has at least k unmatched indices.

For $j \leq k$,

- Let $w(j)$ be the step during which the number of mismatches reaches j^2. (Define $w(0) = 0$.) Thus for $j \geq 1$, step $w(j)$ is a mismatch step that completes block j of γ and $u(w(j)) = j^2$. Clearly the sequence $w(0), w(1), \ldots$ is strictly increasing.

- Define S_j, *phase* j, to be the set of steps $\{w(j-1) + 1, \ldots, w(j)\}$. We refer to phase j as an *odd phase* or an *even phase* depending on whether j is odd or even. For an odd phase we say that x is the *active* string and y is the *inactive* string (since all mismatch steps advance $\mathbf{current}_x$) and for an even phase we say that y is *active* and x is *inactive*.

- Let M_j be the set of pairs added to M during phase j.
- Let $X_{\le j} = \{1, \ldots, c_x(w(j))\}_x$ and $Y_{\le j} = \{1, \ldots, c_y(w(j))\}_y$. Define $X_{\le 0} = Y_{\le 0} = \emptyset$. Thus $X_{\le j}$ (resp. $Y_{\le j}$) is the set of x-indices (resp. y-indices) whose corresponding entries were examined by the end of phase j. Define $X_j = X_{\le j} \backslash X_{\le j-1}$ and $Y_j = Y_{\le j} \backslash Y_{\le j-1}$. It will follow from Proposition 8.6 that the sets X_j and Y_j for $j \le k$ are nonempty.

We prove:

Lemma 8.5. *For every $j \le k$, the number of unmatched indices of M^* in $X_{\le j} \cup Y_{\le j}$ is at least j.*

Setting $j = k$ in the lemma yields that M^* has at least k unmatched indices, completing the proof of Theorem 5.1. So it suffices to prove the lemma.

We start with a short sketch. The proof is by induction on j; assume there are $j-1$ unmatched indices in $X_{\le j-1} \cup Y_{\le j-1}$. We want at least one index in $X_j \cup Y_j$ to be unmatched. Assume j is odd; we focus on the first index of Y_j, called $\mathbf{first}_y(j)$. (If j is even, we focus on the first index of X_j.) If $\mathbf{first}_y(j)$ is unmatched in M^* we're done, so assume $\mathbf{first}_y(j)$ is matched to x-index $\mathbf{first}_y(j) + i$. If $i \ge j$ then Proposition 8.2 implies (without using the induction hypothesis) that $X_{\le j}$ has j unmatched indices. (To show that these unmatched indices are in $X_{\le j}$ requires $\mathbf{first}_y(j) + (j-1) \in X_j$ which can be shown using the fact that block j of γ has $2j-1$ `x`'s. See Proposition 8.7 and Figure 1.) Similarly for $i < j$, Proposition 8.2 implies that $Y_{\le j}$ has j unmatched indices. The remaining case is $i \in [1-j, j-1]$. By induction, it suffices that $X_j \cup Y_j$ has an unmatched index. Phase j has $2j-1$ mismatch steps and (by the final part of Proposition 8.6) at the start of mismatch step m of S_j the offset of **current** is $m - j$. Thus at the start of mismatch step $i + j$, **current** is an unmatched pair $(q + i, q)$ with $q + i \in X_j$ and $q \in Y_j$. By Proposition 8.3, the pairs $(\mathbf{first}_y(j) + i, \mathbf{first}_y(j)) \in M^*$ and $(q + i, q) \notin M^*$ imply an unmatched y-index in $[\mathbf{first}_y(j) + 1, q] \subseteq Y_j$ or x-index in $[\mathbf{first}_y(j) + i + 1, q + i] \subseteq X_j$, to complete the induction.

The details of the above outline are not difficult, but a little tedious. They rely on some detailed claims about the sets X_j and Y_j, which require carefully tracking the values of $\mathbf{current}_x, \mathbf{current}_y$ during a phase. These claims are in Propositions 8.6 and 8.7. The proof of the lemma comes after these propositions.

The following proposition tells how the **current** indices change during a given phase:

Proposition 8.6.

1. $c_y(w(1)) = c_x(w(1)) = |M_1| + 1$ *and* $\delta(w(1)) = 0$.
2. *For odd* $j > 1$,
   (a) $c_y(w(j)) = c_y(w(j-1)) + 1 + |M_j|$
   (b) $c_x(w(j)) = c_x(w(j-1)) + 2j - 2 + |M_j|$
   (c) $\delta(w(j)) = j - 1$
   (d) $\delta(w(j-1) + 1) = -(j-1)$
3. *For even* $j > 1$,
   (a) $c_x(w(j)) = c_x(w(j-1)) + 1 + |M_j|$

(b) $c_y(w(j)) = c_y(w(j-1)) + 2j - 2 + |M_j|$

(c) $\delta(w(j)) = -(j-1)$

(d) $\delta(w(j-1)+1) = j-1$

4. For any $q \in \{-(j-1), j-1\}$, if the $j+q$ mismatch step of S_j occurs at step t then $\delta(t) = q$.

Proof. For the first part: $w(1)$ is the first i such that $x[i] \neq y[i]$. We then have $|M_1| = i-1$ and $c_y(w(1)) = c_x(w(1)) = i$.

For the second part, assume $j > 1$ is odd. Then step $w(j-1)$ is the last mismatch step of the previous y-block. So $c_y(w(j-1)+1) = 1 + c_y(w(j-1))$ and $c_x(w(j-1)+1) = c_x(w(j-1))$. For the steps of S_j prior to $w(j)$ there are $|M_j|$ match steps, and $2j-2$ mismatch steps (since $w(j)$ is the last of $2j-1$ mismatch steps of S_j) and so $c_y(w(j)) - c_y(w(j-1)) = (c_y(w(j)) - c_y(w(j-1)+1)) + (c_y(w(j-1)+1) - c_y(w(j-1)) = |M_j| + 1$ and $c_x(w(j)) - c_x(w(j-1)) = |M_j| + (2j-2)$.

By Proposition 4.1, $\delta(w(j)) = \Delta_\gamma(u(w(j)-1)) = \Delta_\gamma(j^2-1) = j-1$, by Proposition 8.1. Similarly $\delta(w(j-1)+1) = \Delta_\gamma(u(w(j-1))) = \Delta_\gamma((j-1)^2) = -(j-1)$.

The proof of the third part is nearly identical to that of the second part and is omitted.

We prove the final part for j odd, the case of j even is essentially the same. As shown above, the value of δ at the start of the first step in S_j is $1-j$. There are $2j-1$ mismatch steps during the phase. If $s(i)$ is the step during which the i-th mismatch occurs then (by a simple induction)), $\delta(s(i)) = i-j$ and $\delta(s(i)+1) = 1+i-j$. So given $q \in \{1-j, \ldots, j-1\}$, and letting $t = s(j+q)$, we have $\delta(t) = q$. □

This proposition provides the following picture of the movement of **current**. (See Figure 1.) At the start of the first step of an odd phase $j > 1$, $\mathbf{current}_x$ is $j-1$ positions behind $\mathbf{current}_y$ (and this is a local minimum for δ) and during the phase, δ gradually increases so that at the start of the final step of the phase $\mathbf{current}_x$ is $j-1$ positions ahead of $\mathbf{current}_y$. During that final step, $\mathbf{current}_x$ advances by 1 and $\mathbf{current}_y$ stays the same so at the start of the first step of the following (even) phase $j+1$, $\mathbf{current}_x$ is j positions ahead of $\mathbf{current}_y$, and this is a local maximum for δ.

Similarly, at the start of the first step of an even phase j, $\mathbf{current}_x$ is $j-1$ positions ahead of $\mathbf{current}_y$ (and this is a local maximum for δ) and during the phase, δ gradually decreases, so that at the start of the final step of the phase $\mathbf{current}_x$ is $j-1$ positions behind $\mathbf{current}_y$. During that final step, $\mathbf{current}_y$ advances by 1 and $\mathbf{current}_x$ stays the same so at the start of the first step of the following (even) phase $j+1$, $\mathbf{current}_y$ is j positions behind $\mathbf{current}_y$, and this is a local minimum for δ.

Define $\mathbf{last}_x(j) = c_x(w(j))$ and $\mathbf{last}_y(j) = c_y(w(j))$; these are, respectively, the largest indices of $X_{\leq j}$ and $Y_{\leq j}$. Recall that $X_j = X_{\leq j} \backslash X_{\leq j-1}$ and $Y_j = Y_{\leq j} \backslash Y_{\leq j-1}$. From Proposition 8.6 we have that for $j \geq 2$, $\mathbf{last}_x(j) > \mathbf{last}_x(j-1)$ and $\mathbf{last}_y(j) > \mathbf{last}_y(j-1)$ and therefore the sets X_j and Y_j are nonempty. Define $\mathbf{first}_x(j) = \mathbf{last}_x(j-1) + 1$ and $\mathbf{first}_y(j) = \mathbf{last}_y(j-1) + 1$, so that $X_j = \{\mathbf{first}_x(j), \ldots, \mathbf{last}_x(j)\}$ and $Y_j = \{\mathbf{first}_y(j), \ldots, \mathbf{last}_y(j)\}$.

One slightly subtle point: while $(\mathbf{last}_x(j), \mathbf{last}_y(j))$ is the value of **current** at the start of the final step $w(j-1)$ of S_{j-1}, it is **not** true that $(\mathbf{first}_x(j), \mathbf{first}_y(j))$ is the value of **current** at the start of the first step $w(j-1)+1$ of S_j. For j odd we have:

- $c(w(j-1)) = (\mathbf{last}_x(j-1), \mathbf{last}_y(j-1))$
- $c(w(j-1)+1) = (\mathbf{last}_x(j-1), \mathbf{first}_y(j))$, since $w(j-1)$ is a mismatch step of even phase S_{j-1}

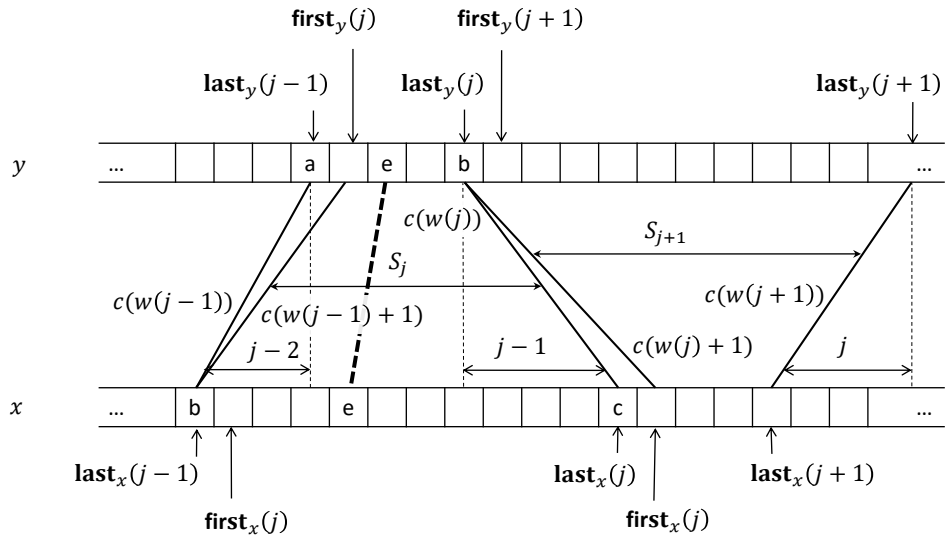

Figure 1: Illustration of a portion of the execution of the algorithm for an odd phase $j > 1$. For $t \in \{w(j-1), w(j-1)+1, w(j), w(j)+1, w(j+1)\}$ marking the beginning and end of phases, pairs $c(t)$ are represented by solid lines. Symbols e are matched in M^*.

- $$c(w(j-1)+2) = \left\{ \begin{array}{ll} (\mathbf{first}_x(j), \mathbf{first}_y(j)), & \text{if } w(j-1)+1 \text{ is a mismatch step} \\ (\mathbf{first}_x(j), \mathbf{first}_y(j)+1), & \text{if } w(j-1)+1 \text{ is a match step.} \end{array} \right\}$$

For j even we have:

- $c(w(j-1)) = (\mathbf{last}_x(j-1), \mathbf{last}_y(j-1))$
- $c(w(j-1)+1) = (\mathbf{first}_x(j), \mathbf{last}_y(j-1))$, since $w(j-1)$ is a mismatch step of odd phase S_{j-1}
- $$c(w(j-1)+2) = \left\{ \begin{array}{ll} (\mathbf{first}_x(j), \mathbf{first}_y(j)), & \text{if } w(j-1)+1 \text{ is a mismatch step} \\ (\mathbf{first}_x(j)+1, \mathbf{first}_y(j)), & \text{if } w(j-1)+1 \text{ is a match step.} \end{array} \right\}$$

For odd phase j, the y-index $\mathbf{first}_y(j)$ plays a central role, and for even phase j, x-index $\mathbf{first}_x(j)$ plays the central role. The following proposition relates various quantities to them.

Proposition 8.7.

1. *For odd $j > 1$,*

(a) $\mathbf{last}_y(j) = \mathbf{first}_y(j) + |M_j|$
(b) $\mathbf{first}_x(j) = \mathbf{first}_y(j) - (j-2)$
(c) $\mathbf{last}_x(j) = \mathbf{first}_y(j) + (j-1) + |M_j|$.

2. *For even $j > 1$,*

(a) $\mathbf{last}_x(j) = \mathbf{first}_x(j) + |M_j|$
(b) $\mathbf{first}_y(j) = \mathbf{first}_x(j) - (j-2)$
(c) $\mathbf{last}_y(j) = \mathbf{first}_x(j) + (j-1) + |M_j|$.

Proof. Since $\mathbf{first}_y(j) = c_y(w(j-1)) + 1$ and $\mathbf{last}_y(j) = c_y(w(j))$ and $\mathbf{first}_x(j) = c_x(w(j-1)) + 1$ and $\mathbf{last}_x(j) = c_x(w(j))$, all of the claimed equalities follow immediately from Proposition 8.6. □

Proof of Lemma 8.5. We proceed by induction on j. Since $j \le k$, A_γ completes block j of γ.

For the base case $j = 1$, if $(1,1) \notin M^*$ then the x-index 1 is unmatched or the y-index 1 is unmatched in M^*, so assume $(1,1) \in M^*$. Since $1 \le k$, A_γ has at least one mismatch step, let this mismatch step be in step i so that $X_{\le 1} = Y_{\le 1} = \{1, \ldots, i\}$. So $(1,1) \in M^*$ and $(i,i) \notin M^*$ so by Proposition 8.3, there is an index in $\{2, \ldots, i\}_x \cup \{2, \ldots, i\}_y \subseteq X_{\le 1} \cup Y_{\le 1}$ that is unmatched, as required.

For the induction step, assume $j > 1$. We'll assume that j is odd; the case that j is even is nearly identical with x and y interchanged, and is omitted. Assume by induction that there are at least $j-1$ indices of $X_{\le j-1} \cup Y_{\le j-1}$ that are unmatched by M^*. It suffices to show that there is an index in $X_j \cup Y_j$ that is unmatched in M^*. We focus on the y-index $\mathbf{first}_y(j)$.

If $\mathbf{first}_y(j)$ is unmatched in M^*, then since $\mathbf{first}_y(j) \in Y_j$, we are done. So assume that $\mathbf{first}_y(j)$ is matched in M^* and let i be such that $(\mathbf{first}_y(j) + i, \mathbf{first}_y(j)) \in M^*$. We divide into cases depending on the value of i. In the first two cases we show directly that $X_{\le j} \cup Y_{\le j}$ has at least j unmatched indices; the induction hypothesis is only used in Case 3.

Case 1. $i \le -j$. Since an x-index $\le \mathbf{first}_y(j) - j$ is matched to the y-index $\mathbf{first}_y(j)$ by Proposition 8.2 there are at least j y-indices from $\{1, \ldots, \mathbf{first}_y(j) - 1\} \subseteq Y_{\le j}$ that are unmatched in M^*.

Case 2. $i \ge j$. Since $\mathbf{first}_y(j)$ is matched to an x-index $\ge \mathbf{first}_y(j) + j$, Proposition 8.2 implies that there are at least j unmatched x-indices in $\{1, \ldots, \mathbf{first}_y(j) + j - 1\}$ and by part 1.c. of Proposition 8.7, $\mathbf{first}_y(j) + j - 1 \le \mathbf{last}_x(j)$ and so the previous set is a subset of $X_{\le j} = \{1, \ldots, \mathbf{last}_x(j)\}$.

Case 3. $-j + 1 \le i \le j - 1$. By the last part of Proposition 8.6 there is a mismatch step $t \le w(j)$ during phase S_j such that $\delta(t) = i$, i.e., $c_x(t) = c_y(t) + i$. So $(\mathbf{first}_y(j) + i, \mathbf{first}_y(j)) \in M^*$ and $(c_y(t) + i, c_y(t)) \notin M^*$ and $\mathbf{first}_y(j) < c_y(t)$ so by Proposition 8.3 there is an unmatched x-index in $\{\mathbf{first}_y(j) + i + 1, c_x(t)\}$ or an unmatched y-index in $\{\mathbf{first}_y(j) + 1, c_y(t)\}$. The first set is a subset of $X_j = \{\mathbf{first}_x(j), \ldots, \mathbf{last}_x(j)\}$ since $\mathbf{first}_y(j) + i + 1 \ge \mathbf{first}_y(j) - (j-2)$ which is $\mathbf{first}_x(j)$ (by Proposition 8.7) and $c_x(t) \le c_x(w(j)) = \mathbf{last}_x(j)$. The second set is a subset of $Y_j = \{\mathbf{first}_y(j), \ldots, \mathbf{last}_y(j)\}$ since $c_y(t) \le c_y(w(j)) = \mathbf{last}_y(j)$. Therefore we have an index in $X_j \cup Y_j$ that is unmatched in M^* as required to complete the induction in this case.

This completes the proof of the Lemma 8.5 and also Theorem 5.1. □

References

[1] Amir Abboud and Arturs Backurs. Towards hardness of approximation for polynomial time problems. In *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, pages 11:1–11:26, 2017.

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

[3] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. *SIAM J. Comput.*, 47(6):2527–2555, 2018. doi:10.1137/16M1061771.

[4] 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 *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388, 2016.

[5] Alex Andoni. Simpler constant-factor approximation to edit distance problems. *Manuscript*, 2018.

[6] 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, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 377–386, 2010.

[7] Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 990–1001. IEEE, 2020. doi:10.1109/FOCS46700.2020.00096.

[8] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 199–204, New York, NY, USA, 2009. ACM.

[9] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, STOC '15, pages 51–58, New York, NY, USA, 2015. ACM.

[10] Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016*, pages 477–488. ACM, 2016. doi:10.1145/2902251.2902304.

[11] Z. Bar-Yossef, T.S. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 550–559, Oct 2004.

[12] Tugkan Batu, Funda Ergün, Joe Kilian, Avner Magen, Sofya Raskhodnikova, Ronitt Rubinfeld, and Rahul Sami. A sublinear algorithm for weakly approximating edit distance. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 316–324, New York, NY, USA, 2003. ACM.

[13] Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 792–801, Philadelphia, PA, USA, 2006. Society for Industrial and Applied Mathematics.

[14] Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 51–60. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.15.

[15] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1170–1189, 2018.

[16] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, Mohammad Taghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and MapReduce (extended version of [15]). 2018.

[17] Joshua Brakensiek and Aviad Rubinstein. Constant-factor approximation of near-linear edit distance in near-linear time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 685–698. ACM, 2020. doi:10.1145/3357713.3384282.

[18] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. *SIAM J. Comput.*, 48(2):481–512, 2019. doi:10.1137/17M112720X.

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

[20] 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 IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990, 2018. doi:10.1109/FOCS.2018.00096.

[21] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 712–725, 2016.

[22] Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for dyck edit distance and RNA folding. In Mikolaj Bojanczyk, Emanuela Merelli, and David P.

Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022*, volume 229 of *LIPIcs*, pages 49:1–49:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.49.

[23] Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. An improved algorithm for the k-dyck edit distance problem. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3650–3669. SIAM, 2022. doi:10.1137/1.9781611977073.144.

[24] Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016.

[25] Ce Jin, Jelani Nelson, and Kewen Wu. An improved sketching algorithm for edit distance. In Markus Bläser and Benjamin Monmege, editors, *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, volume 187 of *LIPIcs*, pages 45:1–45:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.STACS.2021.45.

[26] Tomasz Kociumaka and Barna Saha. Sublinear-time algorithms for computing & embedding gap edit distance. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1168–1179. IEEE, 2020. doi:10.1109/FOCS46700.2020.00112.

[27] Michal Koucký and Michael E. Saks. Constant factor approximations to edit distance on far input pairs in nearly linear time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 699–712. ACM, 2020. doi:10.1145/3357713.3384307.

[28] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, April 1998.

[29] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002. doi:10.1145/505241.505242.

[30] 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.

[31] Barna Saha. The dyck language edit distance problem in near-linear time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 611–620, 2014.

[32] Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 118–135. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.17.

[33] Barna Saha. Fast & space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017*, pages 295–306. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.35.

[34] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[35] Haoyu Zhang and Qin Zhang. Embedjoin: Efficient edit similarity joins via embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*, pages 585–594. ACM, 2017. doi:10.1145/3097983.3098003.