Improved Approximation Algorithms for Dyck Edit Distance and RNA Folding

Debarati Das
Pennsylvania State University, University Park, PA, USA

Tomasz Kociumaka
Max Planck Institute for Informatics, Saarbrücken, Germany

Barna Saha
University of California, San Diego, CA, USA

Abstract

The Dyck language, which consists of well-balanced sequences of parentheses, is one of the most fundamental context-free languages. The Dyck edit distance quantifies the number of edits (character insertions, deletions, and substitutions) required to make a given length-n parenthesis sequence well-balanced. RNA Folding involves a similar problem, where a closing parenthesis can match an opening parenthesis of the same type irrespective of their ordering. For example, in RNA Folding, both `()` and `)(` are valid matches, whereas the Dyck language only allows `()` as a match. Both of these problems have been studied extensively in the literature. Using fast matrix multiplication, it is possible to compute their exact solutions in time $O(n^{2.687})$ (Chi, Duan, Xie, Zhang, STOC'22), and a $(1+\epsilon)$-multiplicative approximation is known with a running time of $\Omega(n^{2.372})$.

The impracticality of fast matrix multiplication often makes combinatorial algorithms much more desirable. Unfortunately, it is known that the problems of (exactly) computing the Dyck edit distance and the folding distance are at least as hard as Boolean matrix multiplication. Thereby, they are unlikely to admit truly subcubic-time combinatorial algorithms. In terms of fast approximation algorithms that are combinatorial in nature, the state of the art for Dyck edit distance is an $O(\log n)$-factor approximation algorithm that runs in near-linear time (Saha, FOCS'14), whereas for RNA Folding only an ϵn-additive approximation in $\tilde{O}(\frac{n^2}{\epsilon})$ time (Saha, FOCS'17) is known.

In this paper, we make substantial improvements to the state of the art for Dyck edit distance (with any number of parenthesis types). We design a constant-factor approximation algorithm that runs in $\tilde{O}(n^{1.971})$ time (the first constant-factor approximation in subquadratic time). Moreover, we develop a $(1+\epsilon)$-factor approximation algorithm running in $\tilde{O}(\frac{n^2}{\epsilon})$ time, which improves upon the earlier additive approximation. Finally, we design a $(3+\epsilon)$-approximation that takes $\tilde{O}(\frac{nd}{\epsilon})$ time, where $d \geq 1$ is an upper bound on the sought distance.

As for RNA folding, for any $s \geq 1$, we design a factor-s approximation algorithm that runs in $O(n + (\frac{n}{s})^3)$ time. To the best of our knowledge, this is the first nontrivial approximation algorithm for RNA Folding that can go below the n^2 barrier. All our algorithms are combinatorial in nature.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis

Keywords and phrases Dyck Edit Distance, RNA Folding, String Algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.49

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2112.05866` [18]

Funding *Tomasz Kociumaka*: Work done while at University of California, Berkeley, supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.
Barna Saha: Partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship.

49th International Colloquium on Automata, Languages, and Programming (ICALP 2022).
Editors: Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff;
Article No. 49; pp. 49:1–49:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The Dyck language is a well-known context-free language consisting of well-balanced sequences of parentheses. Ranging from programming syntaxes to arithmetic and algebraic expressions, environments in LaTeX, and tags in HTML/XML documents – we observe instances of the Dyck language everywhere. For a comprehensive discussion on the Dyck language and context-free grammars; see [23, 26]. Given a sequence x of n parentheses (which may be unbalanced), the *Dyck edit distance problem* asks for the minimum number of edits (character insertions, deletions, and substitutions) needed to make x well-balanced. Interestingly, string edit distance, which is one of the fundamental string similarity measures, can be interpreted as a special case of Dyck edit distance.[1]

A simple dynamic programming computes Dyck edit distance in $O(n^3)$ time. In 2016, after nearly four decades, Bringmann, Grandoni, Saha, and Vassilevska Williams [13] gave the first truly subcubic-time exact algorithm for a more general problem of language edit distance [2]. Very recently, Chi, Duan, Xie, and Zhang [17] provided a faster implementation of the same algorithm. The algorithm uses not-so-practical fast Boolean matrix multiplication, arguably so because computing Dyck edit distance is at least as hard as Boolean matrix multiplication [1], and hence combinatorial truly subcubic-time algorithms are unlikely to exist.

A problem closely related to Dyck edit distance is RNA Folding [31]. Both in RNA Folding and Dyck edit distance, parentheses must match in an uncrossing way. However, in an RNA folding instance, a closing parenthesis can match an opening parenthesis of the same type irrespective of the order of their occurrences. For example, under the RNA Folding distance, both `()` and `)(` are valid matches, whereas the Dyck language only allows `()` as a match. In terms of exact computation, they exhibit the same time complexity [1, 13].[2]

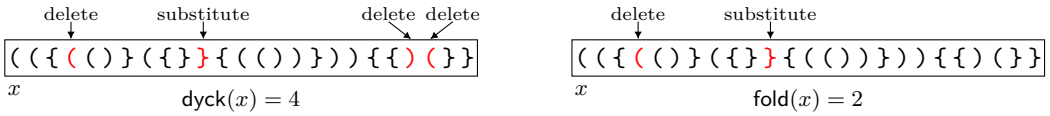

Figure 1 Example of Dyck and folding edit distance.

Can we design fast approximation algorithms for Dyck edit distance and RNA Folding? The first progress on this question for Dyck edit distance was made by Saha [34], who proposed a polylogarithmic-factor approximation algorithm that runs in near-linear time. It is also possible to provide an ϵn-additive approximation for any $\epsilon > 0$ in $\tilde{O}(\frac{n^2}{\epsilon})$ time [36]. However, unless the distance is $O(n)$ and we allow quadratic time, the above algorithm does not provide a constant-factor approximation to Dyck edit distance. This latter result on additive approximation applies to RNA Folding as well. Backurs and Onak [7] showed an exact algorithm for Dyck edit distance that runs in $O(n + d^{16})$ time, which was recently improved by Fried, Golan, Kociumaka, Kopelowitz, Porat, and Starikovskaya [21] to run in $O(n + d^5)$ time (and $\tilde{O}(n + d^{4.783})$ using fast matrix multiplication). Therefore, prior

[1] Given two strings s and t, form a sequence of parentheses by concatenating s, interpreted as a sequence of opening parentheses, and the reverse complement of t, obtained by reversing t and replacing each symbol with the corresponding closing parenthesis, not present in the original alphabet.

[2] In these problems, we are aiming to minimize the number of non-matched parentheses as opposed to maximizing the matched parentheses.

to this work, (i) there was no nontrivial multiplicative approximation for RNA Folding in subquadratic time, and (ii) there was no subquadratic-time constant-factor approximation for Dyck edit distance that would work for the entire distance regime.

Let us contrast this state of affairs with the progress on string edit distance approximation. As mentioned earlier, string edit distance is a special case of Dyck edit distance. Early work [6, 8, 9, 28] on approximating string edit distance resulted in the first near-linear-time polylogarithmic-factor approximation in 2010 by Andoni, Krauthgamer, and Onak [4]. It took another eight years to obtain the first constant-factor approximation of edit distance in subquadratic time [15] (see [11] for a quantum analog). Finally, Andoni and Nosatzki improved the running time to near-linear while maintaining a constant approximation ratio [5]. Using the best result in string edit distance approximation [5], it is possible to improve the approximation factor of [34] to $O(\log n)$. However, designing a constant-factor approximation for Dyck edit distance in subquadratic time remains wide open. RNA Folding, even though conceptually very similar to Dyck edit distance, is incompatible with the algorithm of [34].

Saha's work on Dyck edit distance approximation [34] developed a random walk technique which has later been used for edit distance embedding and document exchange [10, 16]. This random walk allows decomposing a parenthesis sequence into many instances of string edit distance problem. However, this decomposition loses a logarithmic factor in the approximation, raising the question of whether there exists an efficiently computable decomposition with a significantly smaller loss.

Contributions for Dyck Edit Distance.

- **Constant-factor approximation in subquadratic time.** *The main contribution of this paper is the first constant-factor approximation algorithm for Dyck edit distance that runs in truly subquadratic time*, namely $\tilde{O}(n^{1.971})$. (In the interest of simplicity, we did not optimize the exponent in the running time.) We employ and significantly extend the tools previously developed in connection with string edit distance, such as the windowing strategy, window-to-window computation, sparse and dense window decomposition, etc. [11, 15, 22]. These methods are tied to problems involving two or more strings (unlike the Dyck edit distance, which is a single-sequence problem). Given the universality of Dyck edit distance, the tools we developed may lead to further advancements for more generic problems like the language edit distance problem, etc. [13, 35, 36].
  Our main algorithm handles the cases of large and small Dyck edit distance separately.
- **Small Dyck edit distance.** When the Dyck edit distance d is small, we give a $(3+\epsilon)$-approximation algorithm that runs in $\tilde{O}(\frac{nd}{\epsilon})$ time. We can contrast this result with the time complexity of computing the Dyck edit distance exactly, which is $O(n+d^5)$ (combinatorially) and $\tilde{O}(n+d^{4.783})$ (using fast matrix multiplication), obtained in [21]. Nevertheless, even in a hypothetical bast-case scenario that a combinatorial $O(n+d^3)$-time algorithm exists, an $\tilde{O}(nd)$-time algorithm is still faster for all $d \gg \sqrt{n}$.
- **Quadratic-time PTAS.** We also give a $(1+\epsilon)$-approximation algorithm for Dyck edit distance that runs in $\tilde{O}(\frac{n^2}{\epsilon})$ time. This improves upon the previous result of [36] that gets such a result only when $d = \Theta(n)$. The prior $(1+\epsilon)$-approximation algorithm uses fast Boolean matrix multiplication and has super-quadratic running time [35].

Contribution for RNA Folding. For RNA Folding, we are aiming to minimize the number of non-matched characters; we henceforth call this value the *folding distance*. For any $s > 1$, we give a factor-s approximation of the folding distance in time $O(n + (\frac{n}{s})^3)$. This is the first result to our knowledge that goes below the quadratic running time (for $s = \omega(n^{1/3})$). We

remark here that the triangle inequality we proved for Dyck distance (Lemma 2.3) as well as the machinery developed in Section 5 apply to the RNA folding problem equally well. This yields a constant-factor approximation for RNA folding in $\tilde{O}(n^{1.971})$ time when the distance is larger than $n^{0.971}$.

Discussion and Open Problems. The resemblance between Dyck and string edit distance has already been studied in the literature. As mentioned earlier, the decomposition obtained by the random walk technique ensures only an $O(\log n)$ approximation [34]. In this work, instead of reducing the Dyck edit distance to string edit distance, we try to find a direct decomposition of the sequence x into different substrings, where for each substring there is a peer such that they are matched by some optimal alignment (with some error leading to a constant-factor approximation). However, unlike the string counterpart, Dyck edit distance does not have the structural property that if an optimal alignment matches the characters of a substring s_1 with the characters of a substring s_2, then the lengths of s_1 and s_2 are roughly the same (see Figure 2). Thus, in our decomposition, the substrings can have varied lengths. In fact, it turns out that if the Dyck edit distance is truly sublinear (i.e., $n^{1-\epsilon}$), then we need to consider roughly n^{ϵ} different lengths to ensure a constant-factor approximation. We remark that this is one of the barriers in further pushing down the running time from subquadratic to $O(n^{1.6+o(1)})$ (as in [22]) or near-linear. We also note that if an analog of our $\tilde{O}(nd)$-time algorithm can be provided for RNA Folding, then we would also get a constant-factor subquadratic algorithm for RNA folding for all distance regimes.

The Dyck recognition problem has been studied extensively in different models, including the streaming [14, 27, 30] and property testing [3, 19, 32] frameworks. However, neither Dyck edit distance nor RNA Folding admits sublinear-time approximation algorithms. Our algorithm for RNA Folding (which also applies to Dyck edit distance after straightforward adaptations) runs in $O(n + (\frac{n}{s})^3)$ time and requires a linear-time preprocessing step that eliminates pairs of matching adjacent characters, which leaves strongly structured instances. This preprocessing step is currently the main barrier to going in the sublinear-time setting.

1.1 Technical Overview

As input to the Dyck edit distance problem, we are given a string x of length n over an alphabet Σ that consists of two disjoint sets T and $\overline{T}$ of opening and closing parentheses respectively. The task is to compute the Dyck edit distance $\mathsf{dyck}(x)$, defined as the minimum number of parentheses insertions, deletions, and substitutions required to make x well-parenthesized.

Quadratic-time PTAS. The standard $O(n^3)$-time algorithm for Dyck edit distance is a dynamic-programming procedure that computes the distance of each substring of the input string. The bottleneck of this approach is that, to compute the distance of each substring $x(i\,..\,j]$, starting at index $i+1$ and ending at index j, one needs to iterate over decompositions of $x(i\,..\,j]$ into a prefix $x(i\,..\,k]$ and a suffix $x(k\,..\,j]$ for every possible intermediate index $k \in (i\,..\,j)$ (this corresponds to the fact that the concatenation of two well-parenthesized expression is a well-parenthesized expression).[3] We call the index k a *pivot* corresponding to a decomposition. The $\tilde{O}(\frac{n^2}{\epsilon})$-time ϵn additive approximation of [36] reduces the number of considered pivots to $\tilde{O}(\frac{1}{\epsilon})$; thus, $\tilde{O}(\frac{n}{\epsilon d})$ (where $d = \mathsf{dyck}(x)$) different pivots would be

[3] For $i, j \in \mathbb{Z}$, we denote $[i\,..\,j] = \{k \in \mathbb{Z} : i \le k \le j\}$, $[i\,..\,j) = \{k \in \mathbb{Z} : i \le k < j\}$, $(i\,..\,j] = \{k \in \mathbb{Z} : i < k \le j\}$, and $(i\,..\,j) = \{k \in \mathbb{Z} : i < k < j\}$.

necessary for a $(1+\epsilon)$-factor approximation (which is same as an ϵd-additive approximation). On the other hand, a simple $O(n^2 d)$-time algorithm recently developed in [21] is based on a combinatorial observation that $O(d)$ pivots are sufficient after the $O(n)$-time preprocessing from [7, 34]. We start with a brief overview of this algorithm. For any index $i \in [0\,..\,n]$, we define the height of i to be $h(i) = |\{j \in [1\,..\,i] : x[j] \in T\}| - |\{j \in [1\,..\,i] : x[j] \in \overline{T}\}|$, i.e., the difference between the number of opening and closing parentheses in prefix $x[1\,..\,i]$. An index v is called a valley if $h(v-1) > h(v) < h(v+1)$, i.e., $x[v]$ is a closing parenthesis whereas $x[v+1]$ is an opening parenthesis. Backurs and Onak [7] showed a linear-time preprocessing of x that generates another string x' such that $\mathsf{dyck}(x) = \mathsf{dyck}(x')$ and x' has at most $2d$ valleys. Fried, Golan, Kociumaka, Kopelowitz, Porat, and Starikovskaya [21] proved that, without loss of generality, it is enough to consider pivots that are at distance 0 or 1 from a valley (we henceforth denote the set of such pivots by K) plus $O(1)$ pivots next to the boundary of the considered range $(i\,..\,j)$; this observation yields a $O(n^2 d)$-time algorithm.

In Section 3, we provide an algorithm that further restricts the set of pivots K considered for each range $(i\,..\,j)$ and provides a $(1+\epsilon)$-approximation of $\mathsf{dyck}(x)$ in $\tilde{O}(\frac{n^2}{\epsilon})$ time (Theorem 3.2). This is inspired by how Saha [36] considered only $\tilde{O}(\frac{1}{\epsilon})$ pivots out of each range $(i\,..\,j)$. The original argument relies on two observations: that using pivot k' instead of k incurs at most $O(|k-k'|)$ extra edit operations, and that, for an ϵn-additive approximation, we can afford $O(\frac{\epsilon \min(k-i, j-k)}{\log n})$ extra operations when using pivot $k \in (i\,..\,j)$. In our multiplicative approximation, we refine the second observation by replacing $\min(k-i, j-k)$ with $\min(|K \cap (i\,..\,k)|, |K \cap (k\,..\,j)|)$. On the other hand, the first observation is not useful because the set $K \cap (i\,..\,j)$ is already relatively sparse. Thus, instead of restricting each range $(i\,..\,j)$ to use few pivots k, we restrict each pivot k to be used within few ranges $(i\,..\,j)$. This is feasible with respect to the approximation ratio because the costs for $x(i\,..\,j]$ and $x(i'\,..\,j']$ may only differ by $O(|i-i'| + |j-j'|)$, and because the $O(n^2 d)$-time algorithm still considers each pivot $k \in K$ for all ranges $(i\,..\,j)$ containing k (which leaves room for sparsification).

Constant-factor approximation in $\tilde{O}(nd)$ time. Overcoming the $O(n^2)$ barrier with a dynamic-programming approach poses significant challenges: there are $\Theta(n^2)$ substrings to consider and, for $d \geq \sqrt{n}$, this quantity does not decrease (in the worst case) even if we run the preprocessing of [7, 34] and exclude substrings with costs exceeding d. Thus, we artificially restrict the DP states to substrings whose all prefixes have at least as many opening parentheses as closing ones and whose all suffixes have at least as many closing parentheses as opening ones. Surprisingly, as shown in Section 4, this yields a 3-approximation of the original cost. Furthermore, if we additionally require that the number of opening parentheses and the number of closing parentheses across the entire substring are within $2d$ from each other (otherwise, the Dyck edit distance trivially exceeds the threshold), we end up with $O(nd)$ substrings. Reusing the pivot sparsification of Section 3, one can process them in $\tilde{O}(\frac{nd}{\epsilon})$ total time at the cost of increasing the approximation ratio from 3 to $3+\epsilon$.

Constant-factor approximation in $\tilde{O}(n^{1.971})$ time. In Section 5, we exhibit an $\tilde{O}(n^{1.971})$-time algorithm that provides a constant-factor approximation of Dyck edit distance. At a high level, the framework of our algorithm is similar to the three-step procedure of [15] that provides a constant-factor approximation of string edit distance in subquadratic time. Thus, we start with a brief recap of [15], pointing out the major bottlenecks for applying this framework directly to our problem. Given two strings x, y of length n, the algorithm of [15] starts by constructing a set of windows $\mathcal{W}_x$ for x and $\mathcal{W}_y$ for y, where each window is a length-s subinterval of $[1\,..\,n]$, representing a substring of x or y. The motivation behind

this construction is the following: given the edit distances between all pairs of windows from $\mathcal{W}_x$ and $\mathcal{W}_y$, one can compute a constant-factor approximation of the edit distance $\mathsf{ED}(x, y)$ using an $O(\frac{n^2}{s^2})$-time dynamic-programming procedure. The challenge here is that if the edit distances are computed using a trivial dynamic-programming algorithm for all pair of windows from $\mathcal{W}_x$ and $\mathcal{W}_y$, then the total running time becomes quadratic. The key insight of [15] is that, in some favorable situation, one can use random sampling to select a subset of window pairs from $\mathcal{W}_x \times \mathcal{W}_y$ such that evaluating the edit distances of the window pairs in the subset is enough to construct a nearly-optimal alignment of x, y. On the other extreme, instead of computing edit distance for each window pairs explicitly, one can use triangle inequality to get constant-factor approximation of the optimal costs.[4] Several other works have subsequently used this framework to solve related problems [12, 22, 33, 37].

As discussed above, much of the previous work on Dyck edit distance relies on similarities with edit distance, either via black-box reductions (such as the random walk of Saha [34]) or by transferring techniques (e.g., [7, 21] build on top of the $O(n + d^2)$-time algorithm [29] for edit distance). Hence, we try to adapt the framework of [15] to the Dyck setting.

The first challenge is that the Dyck edit distance is defined for a single string, so it is not immediately clear how to formulate the triangle inequality in this setting. However, the embedding of string edit distance into the Dyck edit distance hints a candidate for a metric: a function mapping strings $x, y \in \Sigma^*$ to $\mathsf{dyck}(x\overline{y})$, where $\overline{y}$ is the reverse complement of y (obtained by reversing y and flipping the direction of each parenthesis). This choice turns out to be a valid one: we show (in Lemma 2.3) that any three strings x, y, z satisfy $\mathsf{dyck}(x\overline{z}) \leq \mathsf{dyck}(x\overline{y}) + \mathsf{dyck}(y\overline{z})$, which we dub the triangle inequality for Dyck edit distance. Our proof is based on a subtle inductive argument that reduces the general case to that of $|y| \leq 1$ and $|x\overline{z}| \leq 2$. This base case, in turn, requires some case analysis.

A more serious issue is that the very first step of the algorithms of [11, 15], *window decomposition*, fails for our purposes, and, as discussed below, a workaround poses significant difficulties. To conclude the high-level discussion, we list our two main technical contributions leading to the subquadratic-time constant-factor approximation for Dyck edit distance:

1. We propose a new window decomposition strategy and show that any optimal alignment of x can be approximated by matching the window pairs generated by our strategy.
2. We establish the triangle inequality for Dyck edit distance.

Next, we discuss the limitations of the windowing strategy of [15] and explain how to overcome them. The algorithm of [15] partitions the input strings into fixed-length (overlapping) substrings, estimates the distances between relevant pairs of substrings, and then runs a dynamic-programming procedure to derive a global alignment. Regardless, following the strategy of dimension reduction, one idea could be to partition the input string x into windows $w_1, \ldots, w_\ell \subseteq [1 \,..\, n]$ of length s (where $s = n^{\Theta(1)}$) with the hope, that given the Dyck edit distances for all strings $x[w_i] \circ x[w_j]$ (here, $x[w_i]$ represents the substring of x restricted to the indices in w_i and $\circ$ denotes concatenation), one can use the cubic-time dynamic-programming algorithm to estimate $\mathsf{dyck}(x)$ in time $\tilde{O}(\frac{n^3}{s^3})$. However, this straightforward decomposition fails for the following reasons:

- In case of string edit distance, if an optimal alignment (with cost d) matches $x[i]$ with $y[j]$, then $x[i+1]$ can be matched only with a character of $y[j+1 \,..\, j+d+1]$. Thus, if we consider a window w_1 in x and a window w_2 in y such that $\mathsf{ED}(x[w_1], y[w_2])$ is

[4] The use of triangle inequality was first proposed in [11], where Grover search was used instead of random sampling, resulting in a quantum constant-factor approximation of edit distance in subquadratic time.

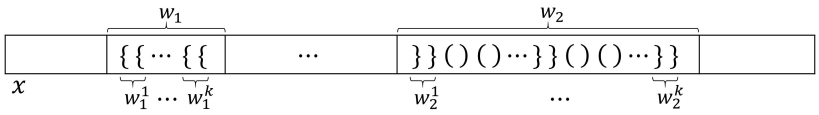

Figure 2 An example showing two very different length substrings can be matched with cost 0.

small, then we can assume $|w_1| \approx |w_2|$. This structural property completely breaks down for Dyck edit distance. For example, the two windows w_1, w_2 in Figure 2 satisfy $\mathsf{dyck}(x[w_1] \circ x[w_2]) = 0$ even though their lengths are very different. This indicates that partitioning x into single-length windows does not suffice.

- To overcome the aforementioned issue, let us assume that we allow variable-length windows. Note that an optimal alignment may match a window w_1 of length s with a window w_2 of length $\gg s$ ($|w_2|$ can be as large as $\Omega(n)$). However, if we allow windows of lengths $\gg s$, then estimating the costs of such large window pairs may be inefficient. One way out could be to subdivide both w_1 and w_2 into smaller windows $w_1^1, w_1^2, \ldots, w_1^k$ and $w_2^1, w_2^2, \ldots, w_2^k$, respectively, and separately compute the cost of each substring $x[w_1^i] \circ x[w_2^j]$. Here, as $|w_1^i|$ and $|w_2^j|$ are not too large, any individual cost can be approximated efficiently. However, since $|w_1^i|$ can be very small (as small as $n^{o(1)}$), the total number of subproblems (window pairs whose cost we evaluate) can explode, and hence the dynamic-programming procedure combining these subproblems may become inefficient.

Thus, for Dyck edit distance, the main challenge is to partition the input string x into variable-length windows which are neither too short (this ensures that the total number of subproblems, i.e., window pairs we evaluate, is not too large, and hence the DP combining them is efficient) nor too long (so that computing the costs of window pairs is efficient as well), and any optimal alignment of x can be approximated by matching these window pairs. Formally, we need to construct a set of windows $\mathcal{J}$, where each window has length at most s (we set this s to be a polynomial in n), $|\mathcal{J}| \approx \frac{n}{s}$, and there exists a subset $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$ such that $\mathcal{S}$ is a consistent window decomposition of $[1\,..\,n]$ (i.e., the windows involved in $\mathcal{S}$ form a decomposition of $[1\,..\,n]$ and the window pairs in $\mathcal{S}$ do not cross) and there is a nearly-optimal alignment that aligns w with w' for each $(w, w') \in \mathcal{S}$. The latter condition is formalized as follows (the construction of $\mathcal{J}$ is parameterized by θ, chosen so that $\theta n \le \mathsf{dyck}(x)$):

▶ **Lemma 1.1.** *There exists a consistent window decomposition $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$ of $[1\,..\,n]$ such that $\sum_{(w,w')\in\mathcal{S}} \mathsf{dyck}(x[w] \circ x[w']) \le \mathsf{dyck}(x) + 8\theta n$.*

The construction of an appropriate family $\mathcal{J}$ and the proof of Lemma 1.1 are among the novelties of our algorithm; this is where our approach significantly differs from [15].

Window Decomposition. Our proof of Lemma 1.1 (given in Section 5.1) follows a two-step strategy. In the first step, independent of the choice of $\mathcal{J}$, the decomposition $\mathcal{S}$ may contain arbitrary window pairs (w, w') with $|w|, |w'| \le s$, but we require $\sum_{(w,w')\in\mathcal{S}} \mathsf{dyck}(x[w] \circ x[w']) = \mathsf{dyck}(x)$ (no approximation allowed) and $|\mathcal{S}| = O(\frac{n}{s})$. In the second step, we locally perturb the endpoints of all windows in $\mathcal{S}$ so that the resulting windows belong to $\mathcal{J}$; this incurs an additive overhead of $O(\theta n)$ on the cost of the consistent window decomposition $\mathcal{S}$.

Our proof for the **first step** inductively constructs a consistent window decomposition of any window $(i_1\,..\,i_2] \subseteq [1\,..\,n]$. In the base case of $|(i_1\,..\,i_2]| \le 2s$, we build a single window pair composed of the two halves of $(i_1\,..\,i_2]$. In the main case, we identify an *outermost* window pair $((i_1\,..\,j_1], (j_2\,..\,i_2])$, with $j_1 \in [i_1\,..\,i_1 + s]$ and $j_2 \in [i_2 - s\,..\,i_2]$, and a *pivot* $p \in [i_1 + s\,..\,i_2 - s]$ so that

$$\mathsf{dyck}(x(i_1\,..\,i_2]) = \mathsf{dyck}(x(j_1\,..\,p]) + \mathsf{dyck}(x(p\,..\,j_2]) + \mathsf{dyck}(x(i_1\,..\,j_1] \circ x(j_2\,..\,i_2]).$$

The appropriate positions j_1, j_2, p can be derived from an optimal alignment of $x(i_1 \,..\, i_2]$:

- $(i_1 \,..\, j_1]$ can be defined as the shortest (possibly empty) prefix of $(i_1 \,..\, i_1 + s]$ containing all positions in $(i_1 \,..\, i_1 + s]$ matched with $(i_2 - s \,..\, i_2]$;
- $(j_2 \,..\, i_2]$ can be defined as the shortest (possibly empty) suffix of $(i_2 - s \,..\, i_2]$ containing all positions in $(i_2 - s \,..\, i_2]$ matched with $(i_1 \,..\, i_1 + s]$;
- $(i_1 + s \,..\, p]$ can be defined as the shortest (possibly empty) prefix of $(i_1 + s \,..\, i_2 - s]$ containing all positions in $(i_1 + s \,..\, i_2 - s]$ matched with $(i_1 \,..\, i_1 + s]$.

The sought decomposition of $(i_1 \,..\, i_2]$ is obtained by inserting $((i_1 \,..\, j_1], (j_2 \,..\, i_2])$ to the union of decompositions of $(j_1 \,..\, p]$ and $(p \,..\, j_2]$ (constructed recursively). It is not hard to prove that this construction satisfies the claims made above. The most subtle argument involves the size of the decomposition. This is because the bound $|\mathcal{S}| = O(\frac{n}{s})$ requires windows of average size $\Theta(s)$, but $(i_1 \,..\, j_1]$ and $(j_2 \,..\, i_2]$ can be arbitrarily short (even empty). Even worse, $(j_1 \,..\, p]$ and $(p \,..\, j_2]$ may also be arbitrarily short. However, we still have $|(i_1 \,..\, p]|, |(p \,..\, i_2]| \geq s$, and this suffices to inductively prove an upper bound of $\max(1, \frac{2(i_2 - i_1)}{s} - 1)$.

As for the **second step** of the proof of Lemma 1.1, we need to specify the choice of $\mathcal{J}$, which is parameterized by θ and s. We simply include in $\mathcal{J}$ all windows $w = (i_1 \,..\, i_2]$ of length at most s whose endpoints i_1, i_2 are both integer multiples of θs (in this overview, we assume for simplicity that $\frac{n}{s}$, θs, and $\frac{1}{\theta}$ are all integers). This way, $|\mathcal{J}| = O(\frac{n}{\theta^2 s})$ (there are $O(\frac{n}{\theta s})$ choices for the starting position and $O(\frac{s}{\theta s})$ choices for the length of a window in $\mathcal{J}$). Moreover, each window of length at most s can be transformed to a window in $\mathcal{J}$ by rounding both endpoints up to the nearest multiple of θs. When performed simultaneously on all windows in $\mathcal{S}$, this perturbation preserves the relative order of the windows, and thus $\mathcal{S}$ remains a consistent window decomposition of $[1 \,..\, n]$. Furthermore, for each window pair (w, w'), the value $\mathsf{dyck}(x[w] \circ x[w'])$ changes by at most $4\theta s$. Given that $|\mathcal{S}| \leq \frac{2n}{s}$, the overall additive overhead does not exceed $8\theta n$.

Two-Level Window Decomposition. If we could estimate the cost of each window pair in $\mathcal{J} \times \mathcal{J}$, this would provide a cost estimation for all window pairs in the unknown set $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$ of Lemma 1.1. Thus, using a dynamic-programming procedure to optimize the cost over consistent decompositions $\tilde{\mathcal{S}} \subseteq \mathcal{J} \times \mathcal{J}$ of $[1 \,..\, n]$, we could approximate $\mathsf{dyck}(x)$.

However, similarly to [15], in order to estimate the cost of each window pair in $\mathcal{J} \times \mathcal{J}$, we further partition each large window into smaller windows and estimate the cost of these smaller window pairs. Thus, given another (smaller) window size parameter, we analogously construct a family $\mathcal{K}$ of variable-sized windows. Adapting the argument behind Lemma 1.1, we can show that, for each window pair $(w, w') \in \mathcal{J} \times \mathcal{J}$, the set $w \cup w'$ admits a consistent window decomposition $\mathcal{S}_{(w,w')} \subseteq \mathcal{K} \times \mathcal{K}$ such that $\sum_{(q,q') \in \mathcal{S}_{(w,w')}} \mathsf{dyck}(x[q] \circ x[q']) \leq \mathsf{dyck}(x[w] \circ x[w']) + O(\theta s)$ (Lemma 5.6 is formulated analogously to Lemma 1.1).

Certifying Window Pairs. With the two-level window decomposition at hand, adapting the remaining two phases of [15] is relatively easy. For this, we design a procedure `CertifyWindowPairs` that finds a cost estimation for selected window pairs in $\mathcal{J} \times \mathcal{J}$ and $\mathcal{K} \times \mathcal{K}$. The procedure shares a similar flavor with the Covering algorithm of [15] and relies on the triangle inequality (Lemma 2.3) discussed above. Its implementation and analysis is provided in the full version [18] only. The main guarantee of `CertifyWindowPairs` is that (with high probability) some of the certified window pairs can be combined to form a consistent window decomposition of $[1 \,..\, n]$ whose cost is $O(\mathsf{dyck}(x) + \theta n)$. Thus, a simple dynamic-programming algorithm (also provided in the full version [18] only) can be used to retrieve a constant-factor approximation of $\mathsf{dyck}(x)$ (recall that $\theta n \leq \mathsf{dyck}(x)$).

Folding Distance. The key difference between the *folding distance* (originating from the RNA folding problem) compared to the Dyck edit distance is that the alphabet is no longer partitioned into the set T of opening parentheses and the set $\overline{T}$ of closing parentheses. In other words, every character c can be matched with its complement $\overline{c}$ regardless of their order in the text. In particular, this means that the notions of heights and valleys are not meaningful anymore. Nevertheless, for any fixed alignment, one can distinguish the unmatched, opening (matched with a character to the right), and closing (matched with a character to the left) characters. Moreover, one can still greedily eliminate substrings of the form $c\overline{c}$ (similarly to the preprocessing of [7]). In any optimal alignment of an instance preprocessed this way, there must be an unmatched character between any two characters matched with each other. Although this reduction does not seem helpful in sparsifying the set of pivots to be considered, it does bring a strong structural property: there is a subset $[1\,..\,n]$ of size $n - O(d)$ (containing all characters matched without edits) which admits a consistent window decomposition into $O(d)$ window pairs (w, w') such that $x[w'] = \overline{x[w]}$ (and thus $\mathsf{fold}(x[w] \circ x[w']) = 0$). The strategy behind our $O(s)$-factor approximation is to sacrifice $O(s)$ boundary characters out of each window pair and, in exchange, make sure that the "closing windows" w' have both endpoints at positions divisible by s. We then use Internal Pattern Matching [24, 25] to efficiently search for "opening windows" w that could match our closing windows. Doing so, we cannot guarantee that the opening windows have their endpoints divisible by s, but we can sparsify the set of candidates so that they start at least s positions apart. This results in $O((\frac{n}{s})^3)$ window pairs to be considered and leads to the overall running time of $O(n + (\frac{n}{s})^3)$. The details are given in the full version [18].

2 Preliminaries

The alphabet Σ consists of two disjoint sets T and $\overline{T}$ of *opening* and *closing* parentheses, respectively, with a bijection $\overline{\cdot} : T \to \overline{T}$ mapping each opening parenthesis to the corresponding closing parenthesis. We extend this mapping to an involution $\overline{\cdot} : T \cup \overline{T} \to T \cup \overline{T}$ and then to an involution $\overline{\cdot} : \Sigma^* \to \Sigma^*$ mapping each string $x[1]x[2]\cdots x[n]$ to its reverse complement $\overline{x[n]}\cdots\overline{x[2]}\,\overline{x[1]}$. Given two strings x, y, we denote their concatenation by xy or $x \circ y$.

The *Dyck* language $\mathsf{Dyck}(\Sigma) \subseteq \Sigma^*$ consists of all well-parenthesized expression over Σ; formally, it can be defined using a context-free grammar whose only non-terminal S admits productions $S \to SS$, $S \to \varnothing$ (empty string), and $S \to aS\overline{a}$ for all $a \in T$.

▶ **Definition 2.1.** *The* Dyck edit distance $\mathsf{dyck}(x)$ *of a string $x \in \Sigma^*$ is the minimum number of character insertions, deletions, and substitutions required to transform x to a string in* $\mathsf{Dyck}(\Sigma)$.

We say that $M \subseteq \{(i, j) \subseteq \mathbb{Z}^2 : i < j\}$ is a *non-crossing matching* if any two distinct pairs $(i, j), (i', j') \in M$ satisfy $i < j < i' < j'$ or $i < i' < j' < j$. Such a matching can also be interpreted as a function $M : \mathbb{Z} \to \mathbb{Z} \cup \{\perp\}$ with $M(i) = j$ if $(i, j) \in M$ or $(j, i) \in M$ for some $j \in \mathbb{Z}$, and $M(i) = \perp$ otherwise.

For a string $x \in \Sigma^n$, the *cost* of a non-crossing matching $M \subseteq [n]^2$ on x (henceforth M is called an *alignment* of x) is defined as $\mathsf{cost}_M(x) = n - 2|M| + \sum_{(i,j)\in M} \mathsf{dyck}(x[i]x[j])$.

The following folklore fact (proved for completeness in the full version [18]), relates the Dyck edit distance with the optimum alignment cost.

▶ **Fact 2.2.** *For every string $x \in \Sigma^*$, the Dyck edit distance* $\mathsf{dyck}(x)$ *is the minimum cost* $\mathsf{cost}_M(x)$ *of an alignment M of x.*

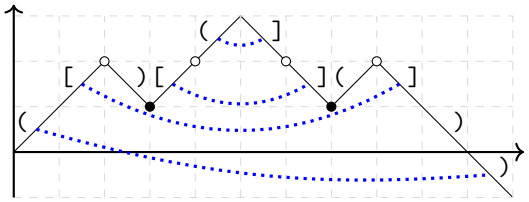

Figure 3 A plot of the height function h for $x = \texttt{([)[(]](])}$. The blue dotted lines represent an alignment $M = \{(1,11),(2,9),(4,7),(5,6)\}$ of cost 4. The valleys $\{3,7\}$ are marked as black circles. The set $K = \{2,3,4,6,7,8\}$ of Fact 2.6 also includes points marked as white circles.

We show that a function mapping $x, y \in \Sigma^*$ to $\mathsf{dyck}(x\overline{y})$ satisfies the triangle inequality.

▶ **Lemma 2.3.** *All strings $x, y, z \in \Sigma^*$ satisfy $\mathsf{dyck}(x\overline{z}) \le \mathsf{dyck}(x\overline{y}y\overline{z}) \le \mathsf{dyck}(x\overline{y}) + \mathsf{dyck}(y\overline{z})$.*

Proof. The second inequality follows from the fact that the Dyck language is closed under concatenations. As for the first inequality, we observe that it suffices to consider $|y| = 1$: the case of $|y| = 0$ is trivial, and the case of $|y| > 1$ can be derived from that of $|y| = 1$ by processing y letter by letter. Now, we proceed by induction on $2\mathsf{dyck}(x\overline{y}y\overline{z}) + |x| + |z|$. If any optimum alignment of $x\overline{y}y\overline{z}$ modifies a character in x or $\overline{z}$, we apply the inductive assumption for an instance (x', y, z') obtained from this modification: $\mathsf{dyck}(x\overline{z}) \le \mathsf{dyck}(x'\overline{z'}) + 1 \le \mathsf{dyck}(x'\overline{y}y\overline{z'}) + 1 = \mathsf{dyck}(x\overline{y}y\overline{z})$. If any optimum alignment of $x\overline{y}y\overline{z}$ matches any two adjacent characters of x, any two adjacent characters of $\overline{z}$, or the first character of x with the last character of $\overline{z}$, we apply the inductive assumption for an instance (x', y, z') obtained by removing these two characters: $\mathsf{dyck}(x\overline{z}) \le \mathsf{dyck}(x'\overline{z'}) \le \mathsf{dyck}(x'\overline{y}y\overline{z'}) = \mathsf{dyck}(x\overline{y}y\overline{z})$. In the remaining case, all characters of x and $\overline{z}$ must be matched to $\overline{y}$ or y, so $|x\overline{z}| \le 2$. If $|x\overline{z}| \le \mathsf{dyck}(x\overline{y}y\overline{z})$, then trivially $\mathsf{dyck}(x\overline{z}) \le |x\overline{z}| \le \mathsf{dyck}(x\overline{y}y\overline{z})$, so we may assume $\mathsf{dyck}(x\overline{y}y\overline{z}) < |x\overline{z}|$. The case of $\mathsf{dyck}(x\overline{y}y\overline{z}) = 0$ and $|x\overline{z}| = 1$ is impossible because only strings of even length belong to the Dyck language. Thus, we may assume that $|x\overline{z}| = 2$ and $\mathsf{dyck}(x\overline{y}y\overline{z}) \le 1$. If $|x| = 2$, then the optimum matching of $x\overline{y}y\overline{z}$ must be $\{(1,4),(2,3)\}$, and the sequence transforming $\mathsf{dyck}(x\overline{y}y\overline{z})$ to a word in $\mathsf{Dyck}(\Sigma)$ must include substituting $\overline{y}$ or y (whichever is an opening parenthesis). In particular, $x[1]$ must be an opening parenthesis, so $\mathsf{dyck}(x\overline{z}) = \mathsf{dyck}(x) \le 1 = \mathsf{dyck}(x\overline{y}y\overline{z})$. If $|\overline{z}| = 2$, then the optimum matching of $x\overline{y}y\overline{z}$ must be $\{(1,4),(2,3)\}$, and the sequence transforming $\mathsf{dyck}(x\overline{y}y\overline{z})$ to a word in $\mathsf{Dyck}(\Sigma)$ must include substituting $\overline{y}$ or y (whichever is a closing parenthesis). In particular, $z[1]$ must be an opening parenthesis, so $\mathsf{dyck}(x\overline{z}) = \mathsf{dyck}(\overline{z}) \le 1 = \mathsf{dyck}(x\overline{y}y\overline{z})$. Finally, if $|x| = |\overline{z}| = 1$, then the optimum matching of $x\overline{y}y\overline{z}$ must be $\{(1,2),(3,4)\}$. If $\mathsf{dyck}(x\overline{y}y\overline{z}) = 0$, then we must have $x = y = z \in T$, so $\mathsf{dyck}(x\overline{z}) = 0 \le \mathsf{dyck}(x\overline{y}y\overline{z})$. Otherwise, $x \in T$ or $z \in T$, so $\mathsf{dyck}(x\overline{z}) \le 1 = \mathsf{dyck}(x\overline{y}y\overline{z})$. ◀

In the remainder of this section, we recall several results from [7, 21] that we then use in our $\tilde{O}_\epsilon(n^2)$-time PTAS (Section 3) and $\tilde{O}_\epsilon(nd)$-time $(3+\epsilon)$-approximation (Section 4).

▶ **Definition 2.4** (Heights). *For a fixed string $x \in \Sigma^n$, the* height *function $h : [0\,..\,n] \to [-n\,..\,n]$ is defined so that $h(i) = |\{j \in [1\,..\,i] : x[j] \in T\}| - |\{j \in [1\,..\,i] : x[j] \in \overline{T}\}|$ for $i \in [0\,..\,n]$.*

▶ **Fact 2.5** ([7]). *There is a linear-time algorithm that, given a string $x \in \Sigma^n$, produces a string $x' \in \Sigma^{\le n}$ such that $\mathsf{dyck}(x) = \mathsf{dyck}(x')$ and x' has at most $2\mathsf{dyck}(x)$* valleys*, i.e., positions $v \in [1\,..\,n)$ such that $h(v-1) > h(v) < h(v+1)$.*

For a fixed string $x \in \Sigma^n$, let us define a function $\mathtt{D}$ such that $\mathtt{D}(i,j) = \mathsf{dyck}(x(i\,..\,j])$ for $i,j \in [0\,..\,n]$ with $i \le j$. Note that $\mathtt{D}(i,i) = 0$ for $i \in [0\,..\,n]$, $\mathtt{D}(i,i+1) = 1$ for $i \in [0\,..\,n)$, and $\mathtt{D}(i,j)$ satisfies the following recursion for $i,j \in [0\,..\,n]$ with $j - i \ge 2$:

$$\mathtt{D}(i,j) = \min \begin{cases} \mathtt{D}(i,k) + \mathtt{D}(k,j) & \text{for } k \in (i\,..\,j), \\ \mathtt{D}(i+1,j-1) + \mathsf{dyck}(x[i+1]x[j]). \end{cases} \tag{1}$$

This yields the classic $O(n^3)$-time algorithm computing $\mathtt{D}(0,n) = \mathsf{dyck}(x)$. The following result, combined with Fact 2.5, improves this time complexity to $O(n + n^2\mathsf{dyck}(x))$.

▶ **Fact 2.6** ([21, Lemma 2.1]). *For a string $x \in \Sigma^n$, let $K \subseteq [0\,..\,n]$ consist of all positions at distance 0 or 1 from a valley. For all $i,j \in [0\,..\,n]$ with $j - i \ge 2$, we have*

$$\mathtt{D}(i,j) = \min \begin{cases} \mathtt{D}(i,k) + \mathtt{D}(k,j) & \text{for } k \in (i\,..\,j) \cap (K \cup \{i+1, i+2, j-1, j-2\}), \\ \mathtt{D}(i+1,j-1) + \mathsf{dyck}(x[i+1]x[j]). \end{cases} \tag{2}$$

▶ **Observation 2.7** ([21, Fact 3.1]). *For all strings $x \in \Sigma^n$ and integers $0 \le i \le k \le j \le n$, we have $h(k) \ge \max(h(i), h(j)) - 2\mathtt{D}(i,j)$. In particular, $|h(i) - h(j)| \le 2\mathtt{D}(i,j)$.*

3 Quadratic-Time PTAS

In this section, we develop an $\tilde{O}(\epsilon^{-1}n^2)$-time algorithm that approximates $\mathsf{dyck}(x)$ within a $(1+\epsilon)$ factor. The starting point of our solution is the dynamic program derived from Fact 2.6. Instead of computing the exact value $\mathtt{D}(i,j) = \mathsf{dyck}(x(i\,..\,j])$, that depends on $\mathtt{D}(i,k) + \mathtt{D}(k,j)$ for all pivots $k \in (i\,..\,j) \cap (K \cup \{i+1, i+2, j-1, j-2\})$, we compute an approximation $\mathtt{AD}(i,j) \approx \mathsf{dyck}(x(i\,..\,j])$ in Algorithm 1 that depends only on $\mathtt{D}(i,k) + \mathtt{D}(k,j)$ for pivots $k \in (i\,..\,j) \cap (K_{i,j} \cup \{i+1, i+2, j-1, j-2\})$, where $K_{i,j}$ consists of $\tau_{i,j}$ leftmost and rightmost elements of $K \cap (i\,..\,j)$. Here, $\tau_{i,j}$ is proportional to the largest power of two dividing both i and j. Formally, we set $\tau_{i,j} := \tau \cdot 2^{\min(\nu(i),\nu(j))}$, where $\tau \ge 2$ is a parameter to be set later and $\nu : \mathbb{Z} \to \mathbb{Z}_{\ge 0} \cup \{\infty\}$ is a function that maps an integer $r \in \mathbb{Z}$ to $\nu(r) := \max\{k \in \mathbb{Z} : 2^k \text{ divides } r\}$, with the convention that $\nu(0) = \infty$.

In the following lemma, we inductively bound the quality of $\mathtt{AD}(i,j)$ as an additive approximation of $\mathtt{D}(i,j)$. In particular, we show that $\mathtt{D}(i,j) \le \mathtt{AD}(i,j) \le \mathtt{D}(i,j) + \frac{8}{\tau}|K|\log|K|$.

▶ **Lemma 3.1.** *If $\tau \ge 2$, then, for each $i,j \in [0\,..\,n]$ with $i \le j$, we have $\mathtt{D}(i,j) \le \mathtt{AD}(i,j) \le \mathtt{D}(i,j) + \frac{8}{\tau}c_{i,j}\log c_{i,j}$, where $c_{i,j} := |K \cap (i\,..\,j)|$, and we assume $0\log 0 = 0$.*

Algorithm 1 Recursive implementation of $\mathtt{AD}(i,j)$.

```
AD(i, j)
    if j = i then return 0;
    if j = i + 1 then return 1;
    c := AD(i + 1, j − 1) + dyck(x[i + 1]x[j]);
    τ_{i,j} := τ · 2^{min(ν(i),ν(j))};
    K_{i,j} := the set of τ_{i,j} smallest and τ_{i,j} largest elements of K ∩ (i .. j);
    foreach k ∈ K_{i,j} ∪ ({i + 1, i + 2, j − 2, j − 1} \ {i, j}) do
        c := min(c, AD(i, k) + AD(k, j));
    return c;
```

Proof. We proceed by induction on $j-i$. For $j-i \le 1$, we have $\mathtt{AD}(i,j) = \mathtt{D}(i,j)$. For $j-i \ge 2$, the lower bound $\mathtt{D}(i,j) \le \mathtt{AD}(i,j)$ follows directly from Fact 2.6. Unless $\mathtt{D}(i,j) = \mathtt{D}(i,k) + \mathtt{D}(k,j)$ for some $k \in (i\,..\,j) \cap K$, the upper bound also follows from Fact 2.6 since $c_{i,j}\log c_{i,j} \ge \max(c_{i+1,j-1}\log c_{i+1,j-1},\ c_{i,k}\log c_{i,k} + c_{k,j}\log c_{k,j})$. Let $r = \min(c_{i,k}, c_{k,j})$ and let i', j' be the smallest and the largest multiples of $2^{\lceil \log((r+1)/\tau)\rceil}$ within $[i\,..\,j]$.

Let us first prove that $k \in K_{i',j'}$. Note that $\tau(i'-i) < \tau 2^{\lceil\log((r+1)/\tau)\rceil} < \tau\cdot 2\cdot\frac{r+1}{\tau} = 2(r+1)$, so $\tau(i'-i) \le 2r$ (because both strict inequalities are between integers). A symmetric argument yields $\tau(j-j') \le 2r$. Due to $\tau \ge 2$, we thus have $i'-i \le r \le c_{i,k} < k-i$ and $j-j' \le r \le c_{j,k} < j-k$, so $k \in (i'\,..\,j')$. Moreover, $\tau_{i',j'} \ge \tau\cdot 2^{\lceil\log((r+1)/\tau)\rceil} \ge r+1 = \min(c_{i,k},c_{k,j})+1 \ge \min(c_{i',k},c_{k,j'})+1$, so $k \in K_{i',j'}$ holds as claimed.

Thus, due to $2r = 2\min(c_{i,k},c_{k,j}) \le c_{i,k}+c_{k,j} \le c_{i,j}$, we have

$$\begin{aligned}
\mathtt{AD}(i,j) &\le (i'-i) + \mathtt{AD}(i',j') + (j-j')\\
&\le \tfrac{2r}{\tau} + \mathtt{AD}(i',k) + \mathtt{AD}(k,j') + \tfrac{2r}{\tau}\\
&\le \mathtt{D}(i',k) + \tfrac{8}{\tau}c_{i',k}\log c_{i',k} + \mathtt{D}(k,j') + \tfrac{8}{\tau}c_{k,j'}\log c_{k,j'} + \tfrac{4r}{\tau}\\
&\le (i'-i) + \mathtt{D}(i,k) + \tfrac{8}{\tau}c_{i,k}\log c_{i,k} + \mathtt{D}(k,j) + (j-j') + \tfrac{8}{\tau}c_{k,j}\log c_{k,j} + \tfrac{4r}{\tau}\\
&\le \mathtt{D}(i,j) + \tfrac{8}{\tau}(c_{i,k}\log c_{i,k} + c_{k,j}\log c_{k,j} + r)\\
&= \mathtt{D}(i,j) + \tfrac{8}{\tau}(\max(c_{i,k},c_{k,j})\log\max(c_{i,k},c_{k,j}) + r\log(2r))\\
&\le \mathtt{D}(i,j) + \tfrac{8}{\tau}(\max(c_{i,k},c_{k,j})\log c_{i,j} + \min(c_{i,k},c_{k,j})\log c_{i,j})\\
&\le \mathtt{D}(i,j) + \tfrac{8}{\tau}c_{i,j}\log c_{i,j}.
\end{aligned}$$

◀

Our final solution simply uses Algorithm 1 with an appropriate choice of the parameter τ and the input string preprocessed using Fact 2.5 so that $|K| = O(\mathsf{dyck}(x))$.

▶ **Theorem 3.2.** *There is an algorithm Dyck-Approx that, given a string $x \in \Sigma^n$ and a parameter $\epsilon \in (0,1)$, in $\tilde{O}(\epsilon^{-1}n^2)$ time computes a value v such that $\mathsf{dyck}(x) \le v \le (1+\epsilon)\mathsf{dyck}(x)$.*

Proof. In the preprocessing, we use Fact 2.5 to guarantee that there are at most $2\mathsf{dyck}(x)$ valleys and thus $|K| \le 6\mathsf{dyck}(x)$. Next, we call $\mathtt{AD}(0,n)$ with $\tau = \lceil 48\epsilon^{-1}\log|K|\rceil$ and an array of size $(n+1)\times(n+1)$ memorizing the outputs of recursive calls. The resulting value satisfies $\mathsf{dyck}(x) \le \mathtt{AD}(0,n) \le \mathsf{dyck}(x) + \frac{8}{\tau}|K|\log|K| \le \mathsf{dyck}(x) + \frac{8}{48\epsilon^{-1}\log|K|}\cdot 6\mathsf{dyck}(x)\cdot\log|K| = (1+\epsilon)\mathsf{dyck}(x)$ by Lemma 3.1. The running time is proportional to

$$\begin{aligned}
n^2\sum_{i=0}^{n}\sum_{j=i+2}^{n}\tau_{i,j} &\le n^2 + \sum_{i=0}^{n}\sum_{j=i+2}^{n}\tau 2^{\nu(j)} \le n^2 + n\tau\sum_{j=2}^{n}2^{\nu(j)} \le n^2 + n\tau\sum_{\nu=0}^{\lfloor\log n\rfloor}\left\lfloor\frac{n}{2^\nu}\right\rfloor 2^\nu\\
&= O(n^2\tau\log n) = O(\epsilon^{-1}n^2\log^2 n) = \tilde{O}(\epsilon^{-1}n^2).
\end{aligned}$$

◀

4 Constant-Factor Approximation for Small Distances

In this section, we speed up the algorithm of Section 3 at the cost of increasing the approximation ratio from $1+\epsilon$ to $3+\epsilon$. The key idea behind our solution is to re-use the DP of Fact 2.6 and Algorithm 1 with an extra constraint that the transition from (i,j) to $(i+1,j-1)$ (which corresponds to adding $(i+1,j)$ to the alignment M, i.e., matching $x[i+1]$ with $x[j]$) is forbidden if there is a deep valley within $(i\,..\,j)$. This condition is expressed in terms of the following function:

▶ **Definition 4.1.** *For a fixed string $x \in \Sigma^n$ and $i,j \in [0\,..\,n]$ with $i \le j$, let $h(i,j) = \min_{k=i}^{j} h(k)$.*

Namely, we require that $h(i+1, j-1) > h(i,j)$ holds for all $(i+1, j) \in M$. For example, in the alignment M of Figure 3, $(2, 9) \in M$ violates this condition due to $h(1, 9) = 1 = h(2, 8)$, whereas the remaining pairs satisfy this condition. Formally, we transform the recursion of Fact 2.6 into the following one, specified through a function $\mathtt{GD} : [0\,..\,n]^2 \to [0\,..\,n]$ such that $\mathtt{GD}(i, i) = 0$ for $i \in [0\,..\,n]$, $\mathtt{GD}(i, i+1) = 1$ for $i \in [0\,..\,n)$, and, for all $i, j \in [0\,..\,n]$ with $j - i \geq 2$:

$$\mathtt{GD}(i,j) = \min \begin{cases} \mathtt{GD}(i,k) + \mathtt{GD}(k,j) & \text{for } k \in (i\,..\,j) \cap (K \cup \{i+1, i+2, j-2, j-1\}), \\ \mathtt{GD}(i+1, j-1) + \mathsf{dyck}(x[i+1]x[j]) & \text{if } h(i+1, j-1) > h(i,j). \end{cases}$$

Somewhat surprisingly, this significant limitation on the allowed alignments M incurs no more that a factor-3 loss in optimum alignment cost. Specifically, if we take an arbitrary alignment M of x and remove all pairs $(i+1, j)$ with $h(i+1, j-1) = h(i,j)$, the resulting alignment M' satisfies $\mathsf{cost}_{M'}(x) \leq 3\mathsf{cost}_M(x)$. This can be proved by induction on the structure of M using a potential function $h(i) + h(j) - 2h(i,j)$ as a "budget" for future deletions of matched pairs. Nevertheless, the proof of the following lemma operates directly on $\mathtt{D}$ and $\mathtt{GD}$.

▶ **Lemma 4.2.** *Let $x \in \Sigma^n$. For all $i, j \in [0\,..\,n]$ with $i \leq j$, we have $\mathtt{D}(i,j) \leq \mathtt{GD}(i,j) \leq 3\mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j)$.*

Proof. We proceed by induction on $j - i$. The lower bound holds trivially. As for the upper bound, we consider several cases:

- $j = i$. In this case, $\mathtt{GD}(i,j) = 0 = 3 \cdot 0 - h(i) - h(i) + 2h(i) = \mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j)$.
- $j = i + 1$. In this case, $\mathtt{GD}(i,j) = 1 < 2 = 3 \cdot 1 - h(i) - h(i+1) + 2\min(h(i), h(i+1)) = \mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j)$.
- $\mathtt{D}(i,j) = \mathtt{D}(i,k) + \mathtt{D}(k,j)$ for some $k \in (i\,..\,j) \cap (K \cup \{i+1, i+2, j-2, j-1\})$. Then,

$$\begin{aligned} \mathtt{GD}(i,j) &\leq \mathtt{GD}(i,k) + \mathtt{GD}(k,j) \\ &\leq 3\mathtt{D}(i,k) - h(i) - h(k) + 2h(i,k) + 3\mathtt{D}(k,j) - h(k) - h(j) + 2h(k,j) \\ &= 3\mathtt{D}(i,j) - h(i) - h(j) - 2h(k) + 2\min(h(i,k), h(k,j)) + 2\max(h(i,k), h(k,j)) \\ &\leq 3\mathtt{D}(i,j) - h(i) - h(j) - 2h(k) + 2h(i,j) + 2h(k) \\ &= 3\mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j). \end{aligned}$$

- $\mathtt{D}(i,j) = \mathtt{D}(i+1, j-1) + \mathsf{dyck}(x[i+1]x[j])$ and $h(i+1, j-1) = h(i,j) + 1$. Then,

$$\begin{aligned} \mathtt{GD}(i,j) &\leq \mathtt{GD}(i+1, j-1) + \mathsf{dyck}(x[i+1]x[j]) \\ &\leq 3\mathtt{D}(i+1, j-1) - h(i+1) - h(j-1) + 2h(i+1, j-1) + \mathsf{dyck}(x[i+1]x[j]) \\ &= 3\mathtt{D}(i,j) - h(i+1) - h(j-1) + 2h(i,j) + 2 - 2\mathsf{dyck}(x[i+1]x[j]) \\ &\leq 3\mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j) \end{aligned}$$

  because $2\mathsf{dyck}(x[i+1]x[j]) \geq 2 + h(i) - h(i+1) + h(j) - h(j-1)$.
- $\mathtt{D}(i,j) = \mathtt{D}(i+1, j-1) + \mathsf{dyck}(x[i+1]x[j])$ and $h(i+1, j-1) = h(i,j)$. Then,

$$\begin{aligned} \mathtt{GD}(i,j) &\leq \mathtt{GD}(i, i+1) + \mathtt{GD}(i+1, j-1) + \mathtt{GD}(j-1, j) \\ &= \mathtt{GD}(i+1, j-1) + 2 \\ &\leq 3\mathtt{D}(i+1, j-1) - h(i+1) - h(j-1) + 2h(i+1, j-1) + 2 \\ &= 3\mathtt{D}(i,j) - h(i+1) - h(j+1) + 2h(i,j) - 3\mathsf{dyck}(x[i+1]x[j]) + 2 \\ &\leq 3\mathtt{D}(i,j) - h(i) - h(j) + 2h(i,j) \end{aligned}$$

  because $3\mathsf{dyck}(x[i+1]x[j]) \geq 2\mathsf{dyck}(x[i+1]x[j]) \geq 2 + h(i) - h(i+1) + h(j) - h(j-1)$. ◀

Next, we derive a property of GD that allows for a speedup compared to D. Recall that GD forbids the transition from (i,j) to $(i+1,j-1)$ if $h(i+1,j-1)=h(i,j)$. We further show that, in this case, it suffices to consider one specific pivot while computing $\mathtt{GD}(i,j)$ (specifically, the pivot of minimum height, with ties resolved arbitrarily; see Fact 4.3). Later on (in the proof of Theorem 4.6), we argue that, after the preprocessing of Fact 2.5, there are only $O(nd)$ pairs (i,j) for which $h(i+1,j-1)>h(i,j)$ yet $\mathtt{D}(i,j)\le d$.

▶ **Fact 4.3.** *Let $x\in\Sigma^n$ and let $i,j\in[0\,..\,n]$ with $j-i\ge 2$ and $h(i+1,j-1)=h(i,j)$. Then, every $k^*\in(i\,..\,j)$ with $h(k^*)=h(i,j)$ satisfies $\mathtt{GD}(i,j)=\mathtt{GD}(i,k^*)+\mathtt{GD}(k^*,j)$.*

Proof. We proceed by induction on $j-i$. Fix $k\in(i\,..\,j)\cap(K\cup\{i+1,i+2,j-2,j-1\})$ such that $\mathtt{GD}(i,j)=\mathtt{GD}(i,k)+\mathtt{GD}(k,j)$. If $k=k^*$, then the claim is trivial. Thus, by symmetry, we assume without loss of generality that $k^*\in(i\,..\,k)$. In particular, this means that $k-i\ge 2$ and $h(i+1,k-1)=h(k^*)=h(i,k)$, Consequently, by the inductive assumption, $\mathtt{GD}(i,j)=\mathtt{GD}(i,k)+\mathtt{GD}(k,j)=\mathtt{GD}(i,k^*)+\mathtt{GD}(k^*,k)+\mathtt{GD}(k,j)\ge\mathtt{GD}(i,k^*)+\mathtt{GD}(k^*,j)\ge\mathtt{GD}(i,j)$, i.e., $\mathtt{GD}(i,j)=\mathtt{GD}(i,k^*)+\mathtt{GD}(k^*,j)$ holds as claimed. Here, the first inequality holds because $k\in(k^*\,..\,j)\cap(K\cup\{k^*+1,k^*+2,j-2,j-1\})$, whereas the second one is due to $k^*\in K$ (because k^* is a valley). ◀

Our approximation algorithm (implemented as Algorithm 2) computes AGD that approximates GD in the same way AD approximates D in Algorithm 1. The only difference is that we use Observation 2.7 and Fact 4.3 (and the definition of GD) to prune some states and transitions. For each of the remaining states, the algorithm computes a value $\mathtt{AGD}(i,j)\approx\mathtt{GD}(i,j)$. If $h(i,j)=h(i+1,j-1)$, then Algorithm 2 relies on Fact 4.3 and considers the smallest index $k\in(i\,..\,j)$ with $h(k)=h(i,j)$ as the sole potential pivot, i.e, it returns $\mathtt{AGD}(i,k)+\mathtt{AGD}(k,j)$. If $h(i,j)<h(i+1,j-1)$, then Algorithm 2 mimics Algorithm 1.

The analysis of the approximation ratio of Algorithm 2 resembles that of Algorithm 1.

▶ **Lemma 4.4.** *If $\tau\ge 2$, then, for each $i,j\in[0\,..\,n]$ with $i\le j$, we have $\mathtt{GD}(i,j)\le\mathtt{AGD}(i,j)$ and, if $\mathtt{GD}(i,j)\le d$, we further have $\mathtt{AGD}(i,j)\le\mathtt{GD}(i,j)+\frac{8}{\tau}c_{i,j}\log c_{i,j}$, where $c_{i,j}:=|K\cap(i\,..\,j)|$, and we assume $0\log 0=0$.*

Proof. As for the upper bound, we proceed by induction on $j-i$. For $j-i\le 1$, we have $\mathtt{AGD}(i,j)=\mathtt{GD}(i,j)$. For $j-i\ge 2$, the lower bound $\mathtt{GD}(i,j)\le\mathtt{AGD}(i,j)$ follows directly form the definitions of AGD and GD. If $h(i,j)<\max(h(i),h(j))-2d$, then the upper bound follows

Algorithm 2 Recursive implementation of $\mathtt{AGD}(i,j)$.

```
1  AGD(i, j)
2      if j = i then return 0;
3      if j = i + 1 then return 1;
4      if h(i, j) < max(h(i), h(j)) − 2d then return ∞;
5      if h(i, j) = h(i + 1, j − 1) then
6          Select the smallest k ∈ (i .. j) such that h(k) = h(i, j);
7          return AGD(i, k) + AGD(k, j);
8      c := AGD(i + 1, j − 1) + dyck(x[i + 1]x[j]);
9      τ_{i,j} := τ · 2^{min(ν(i),ν(j))};
10     K_{i,j} := the set of τ_{i,j} smallest and τ_{i,j} largest elements of K ∩ (i .. j);
11     foreach k ∈ K_{i,j} ∪ ({i + 1, i + 2, j − 2, j − 1} \ {i, j}) do
12         c := min(c, AGD(i, k) + AGD(k, j));
13     return c;
```

from Observation 2.7 and Lemma 4.2. If $h(i,j) = h(i+1,j-1)$, then the upper bound follows form Fact 4.3 because $c_{i,k}\log c_{i,k} + c_{k,j}\log c_{k,j} \le c_{i,j}\log c_{i,j}$. Otherwise, the upper bound follows directly from the definitions of $\texttt{AGD}$ and $\texttt{GD}$ unless $\texttt{GD}(i,j) = \texttt{GD}(i,k) + \texttt{GD}(k,j)$ for some $k \in (i\,..\,j) \cap K$. Let $r = \min(c_{i,k}, c_{k,j})$ and let i', j' be the smallest and the largest multiple of $2^{\lceil\log((r+1)/\tau)\rceil}$ within $[i\,..\,j]$.

Let us next prove that $k \in K_{i',j'}$. Note that $\tau(i'-i) < \tau 2^{\lceil\log((r+1)/\tau)\rceil} < \tau\cdot 2\cdot\frac{r+1}{\tau} = 2(r+1)$, so $\tau(i'-i) \le 2r$ (because both strict inequalities are between integers). A symmetric argument yields $\tau(j-j') \le 2r$. Due to $\tau \ge 2$, we thus have $i' - i \le r \le c_{i,k} < k - i$ and $j - j' \le r \le c_{j,k} < j - k$, so $k \in (i'\,..\,j')$. Moreover, $\tau_{i',j'} \ge \tau\cdot 2^{\lceil\log((r+1)/\tau)\rceil} \ge r+1 = \min(c_{i,k},c_{k,j})+1 \ge \min(c_{i',k},c_{k,j'})+1$, so $k \in K_{i',j'}$ holds as claimed.

Thus, due to $2r = 2\min(c_{i,k},c_{k,j}) \le c_{i,k}+c_{k,j} \le c_{i,j}$, we have

$$\begin{aligned}
\texttt{AGD}(i,j) &\le (i'-i) + \texttt{AGD}(i',j') + (j-j')\\
&\le \tfrac{2r}{\tau} + \texttt{AGD}(i',k) + \texttt{AGD}(k,j') + \tfrac{2r}{\tau}\\
&\le \texttt{GD}(i',k) + \tfrac{8}{\tau}c_{i',k}\log c_{i',k} + \texttt{GD}(k,j') + \tfrac{8}{\tau}c_{k,j'}\log c_{k,j'} + \tfrac{4r}{\tau}\\
&\le (i'-i) + \texttt{GD}(i,k) + \tfrac{8}{\tau}c_{i,k}\log c_{i,k} + \texttt{GD}(k,j) + (j-j') + \tfrac{8}{\tau}c_{k,j}\log c_{k,j} + \tfrac{4r}{\tau}\\
&\le \texttt{GD}(i,j) + \tfrac{8}{\tau}(c_{i,k}\log c_{i,k} + c_{k,j}\log c_{k,j} + r)\\
&= \texttt{GD}(i,j) + \tfrac{8}{\tau}(\max(c_{i,k},c_{k,j})\log\max(c_{i,k},c_{k,j}) + r\log(2r))\\
&\le \texttt{GD}(i,j) + \tfrac{8}{\tau}(\max(c_{i,k},c_{k,j})\log c_{i,j} + \min(c_{i,k},c_{k,j})\log c_{i,j})\\
&\le \texttt{GD}(i,j) + \tfrac{8}{\tau}c_{i,j}\log c_{i,j}.
\end{aligned}$$

◀

On the other hand, the complexity analysis is not as simple as in Section 3: it involves a charging argument bounding the number of states processed using the insight of Fact 4.3.

▶ **Proposition 4.5.** *There is an algorithm that, given a string $x \in \Sigma^n$, a threshold $d \in [1\,..\,n]$, and a parameter $\epsilon \in (0,1)$, in $\tilde{O}(\epsilon^{-1}nd)$ time reports that $\texttt{GD}(0,n) > d$ or outputs a value v such that $\texttt{GD}(0,n) \le v \le (1+\epsilon)\texttt{GD}(0,n)$.*

Proof. In the preprocessing, we use Fact 2.5 to guarantee that there are at most $2\mathsf{dyck}(x)$ valleys and thus $|K| \le 6\mathsf{dyck}(x)$. Then, we construct a data structure that, given $i,j \in [0\,..\,n]$, reports the smallest $k \in [i\,..\,j]$ such that $h(k) = h(i,j)$ [20]. Finally, we run $\texttt{AGD}(0,n)$ with $\tau = \lceil 48\epsilon^{-1}\log|K|\rceil$ and memoization of the results of recursive calls. By Lemma 4.4, the returned value satisfies $\texttt{GD}(0,n) \le \texttt{AGD}(0,n) \le \texttt{GD}(0,n) + \frac{8}{\tau}|K|\log|K| \le (1+\epsilon)\texttt{GD}(0,n)$.

The running time analysis is more complex than in the proof of Theorem 3.2. We say that a call $\texttt{AGD}(i,j)$ is *hard* if it reaches Line 8, *easy* if it terminates at Line 4 or Line 7, and *trivial* otherwise. Observe that the total cost of trivial calls is $\tilde{O}(n)$. Moreover, the cost of each easy call is $\tilde{O}(1)$ plus the cost of the two calls made in Line 7, but the call $\texttt{AGD}(i,k)$ is never easy (it can be hard or trivial). This is because the choice of k as the smallest index in $(i\,..\,j)$ with $h(k) = h(i,j)$ guarantees that either $k = i+1$ or $h(i+1,k-1) > h(k) = h(i,k) = h(i,j) \ge \max(h(i),h(j)) - 2d \ge h(i) - 2d = \max(h(i),h(k)) - 2d$. Consequently, the cost of each easy call can be charged to its parent or sibling (which is hard or trivial), and it suffices to bound the total running time of hard calls. By symmetry, we only bound the cost of hard calls $\texttt{AGD}(i,j)$ with $h(i) \ge h(j)$. We then observe that if $h(i) \ge h(j) = h(j')$ and $i > j > j'$, then $\texttt{AGD}(i,j')$ is easy. Consequently, there are at most $4d+1$ hard calls per i. The cost of each hard call $\texttt{AGD}(i,j)$ is $\tilde{O}(\tau_{i,j}) = \tilde{O}(\tau 2^{\nu(i)})$, for a total of $\tilde{O}(d\tau\sum_{i=0}^{n} 2^{\nu(i)}) = \tilde{O}(\epsilon^{-1}nd)$. ◀

▶ **Theorem 4.6.** *There is an algorithm that, given a string $x \in \Sigma^n$, a threshold $d \in [1\,..\,n]$, and a parameter $\epsilon \in (0,1)$, in $\tilde{O}(\epsilon^{-1}nd)$ time reports that $\mathsf{dyck}(x) > d$ or outputs a value v such that $\mathsf{dyck}(x) \le v \le (3+\epsilon)\mathsf{dyck}(x)$.*

Proof. We apply Proposition 4.5 with adjusted d (three times larger) and ϵ (three times smaller). The correctness follows from Lemma 4.2. ◀

5 Constant-Factor Approximation in Subquadratic Time

In this section, we provide an algorithm that, given a string $x \in \Sigma^*$, computes constant-factor approximation of $\mathsf{dyck}(x)$ in subquadratic time. Formally, we show the following:

▶ **Theorem 5.1.** *There exist a randomized algorithm that, given a string $x \in \Sigma^n$, in $\tilde{O}(n^{67/34}) = O(n^{1.971})$ time, outputs a value v such that $\mathsf{dyck}(x) \le v \le 41 \cdot \mathsf{dyck}(x)$ holds with probability at least $1 - n^{-9}$.*

Instead of directly proving the above theorem, we develop the following result:

▶ **Theorem 5.2.** *There exist a constant C and a randomized algorithm that, given a string $x \in \Sigma^n$, in time $\tilde{O}(n^{67/34})$, outputs a value v such that $\mathsf{dyck}(x) \le v \le 40 \cdot \mathsf{dyck}(x) + Cn^{33/34}$ holds with probability at least $1 - n^{-9}$.*

Proof of Theorem 5.1 from Theorem 5.2. First, run the algorithm of Theorem 4.6 with $\epsilon = 1$ and $d = \lceil Cn^{33/34} \rceil$, where C is the constant of Theorem 5.2, this procedure takes $\tilde{O}(n^{67/34})$ time and either outputs a 4-approximation of $\mathsf{dyck}(x)$ or reports that $\mathsf{dyck}(x) > d$. In the latter case, run the algorithm of Theorem 5.2 and return the resulting value v. Then, $\mathsf{dyck}(x) \le v \le 40 \cdot \mathsf{dyck}(x) + Cn^{33/34} \le 41 \cdot \mathsf{dyck}(x)$ holds with probability at least $1 - n^{-9}$. ◀

The rest of the section is devoted to prove Theorem 5.2.

5.1 Window Decomposition

Let us fix a string $x \in \Sigma^n$. For all integers $0 \le i_1 \le i_2 \le n$, we define a *window* $w := (i_1 \,..\, i_2]$ with *endpoints* $b(w) := i_1$, $e(w) := i_2$ and with *length* $|w| := i_2 - i_1$. We distinguish $n + 1$ distinct *empty* windows $(i \,..\, i]$ for $i \in [0 \,..\, n]$. For $w = (i_1 \,..\, i_2]$, we denote $x[w] := x(i_1 \,..\, i_2]$.

A *window pair* is a pair of windows (w, w'), and a weighed window pair is a triple (w, w', c) such that (w, w') is a window pair and $c \in \mathbb{R}_{\ge 0}$ is a weight. The *cost* of a window pair (w, w') is $\mathsf{dyck}(x[w] \circ x[w'])$, and a weighted window pair (w, w', c) is *certified* if $c \ge \mathsf{dyck}(x[w] \circ x[w'])$.

▶ **Definition 5.3.** *A set $\{(w_1, w_1'), \ldots, (w_\ell, w_\ell')\}$ of window pairs is a* consistent decomposition *of $\bigcup_{i=1}^{\ell}(w_i \cup w_i')$ if the 2ℓ windows are disjoint and $\{(b(w_i), b(w_i')) : i \in [1 \,..\, \ell]\}$ forms a non-crossing matching. We also lift this definition to sets of weighted window pairs.*

For a consistent decomposition $\mathcal{S}$, we write $\mathsf{dyck}(\mathcal{S}) := \sum_{(w,w') \in \mathcal{S}} \mathsf{dyck}(x[w] \circ x[w'])$ to denote the total cost of windows pairs in $\mathcal{S}$. Observe that if $\mathcal{S}$ is a consistent decomposition of $[1 \,..\, n]$, then $\mathsf{dyck}(\mathcal{S}) \ge \mathsf{dyck}(x)$.

Our first goal is to prove that, for every $s \in [1 \,..\, n]$, there exists a consistent decomposition $\mathcal{S}$ of $[1 \,..\, n]$ such that $\mathsf{dyck}(\mathcal{S}) = \mathsf{dyck}(x)$, $|\mathcal{S}| = O(\frac{n}{s})$, and each window in $\mathcal{S}$ is of length at most s. For this, we inductively construct a consistent window decomposition of an arbitrary interval $(i_1 \,..\, i_2] \subseteq [1 \,..\, n]$ specified as follows (recall that $\mathtt{D}(i_1, i_2)$ denotes $\mathsf{dyck}(x(i_1 \,..\, i_2])$):

▶ **Lemma 5.4.** *Let x be a string of length n and let $s \in [1 \,..\, n]$. For every interval $(i_1 \,..\, i_2] \subseteq [1 \,..\, n]$, there exists a consistent decomposition $\mathtt{Dec}(i_1, i_2)$ of $(i_1 \,..\, i_2]$ such that:*

1. *each window pair $(w, w') \in \mathtt{Dec}(i_1, i_2)$ satisfies $|w|, |w'| \le s$,*
2. *$\mathsf{dyck}(\mathtt{Dec}(i_1, i_2)) = \mathtt{D}(i_1, i_2)$, and*
3. *$|\mathtt{Dec}(i_1, i_2)| \le \max(1, \frac{2(i_2 - i_1)}{s} - 1)$.*

Proof. If $|(i_1 \,..\, i_2]| \le 2s$, we return $\mathtt{Dec}(i_1, i_2) := \{((i_1 \,..\, \lfloor \frac{i_1 + i_2}{2} \rfloor], (\lfloor \frac{i_1 + i_2}{2} \rfloor \,..\, i_2])\}$. In this simple base case, all the claims hold trivially.

In the main case, we grow the outermost window pair $((i_1\,..\,j_1],(j_2\,..\,i_2])$, starting with empty windows and maintaining two invariants: $|(i_1\,..\,j_1]|,|(j_2\,..\,i_2]| \le s$ and $\mathtt{D}(i_1,i_2) = \mathtt{D}(j_1,j_2) + \mathsf{dyck}(x(i_1\,..\,j_1]\circ x(j_2\,..\,i_2])$. Once there exists $p \in [i_1+s\,..\,i_2-s]$ with $\mathtt{D}(j_1,j_2) = \mathtt{D}(j_1,p)+\mathtt{D}(p,j_2)$ (in particular, this holds when $|(i_1\,..\,j_1]| = s$ or $|(j_2\,..\,i_2]| = s$), we terminate the process and return $\mathtt{Dec}(i_1,i_2) := \{((i_1\,..\,j_1],(j_2\,..\,i_2])\} \cup \mathtt{Dec}(j_1,p) \cup \mathtt{Dec}(p,j_2)$. By the inductive hypothesis, $\mathtt{Dec}(j_1,p), \mathtt{Dec}(p,j_2)$ satisfy all the claimed conditions. In particular, they form consistent decompositions of $(j_1\,..\,p]$ and $(p\,..\,j_2]$, respectively, and thus $\mathtt{Dec}(i_1,i_2)$ forms a consistent decomposition of $(i_1\,..\,i_2]$. By the first invariant, each window in $\mathtt{Dec}(i_1,i_2)$ is of length at most s. The second invariant and the definition of p yield

$$\begin{aligned}\mathtt{D}(i_1,i_2) &= \mathtt{D}(j_1,p) + \mathtt{D}(p,j_2) + \mathsf{dyck}(x(i_1\,..\,j_1]\circ x(j_2\,..\,i_2])\\ &= \mathsf{dyck}(\mathtt{Dec}(j_1,p)) + \mathsf{dyck}(\mathtt{Dec}(p,j_2)) + \mathsf{dyck}(x(i_1\,..\,j_1]\circ x(j_2\,..\,i_2])\\ &= \mathsf{dyck}(\mathtt{Dec}(i_1,i_2)).\end{aligned}$$

The choice of $p \in [i_1+s\,..\,i_2-s]$ further gives

$$\begin{aligned}&|\mathtt{Dec}(i_1,i_2)| \le 1+|\mathtt{Dec}(j_1,p)|+|\mathtt{Dec}(p,j_2)| \le 1+\max(1,\tfrac{2(p-j_1)}{s}-1)+\max(1,\tfrac{2(j_2-p)}{s}-1)\\ &\le 1+\max(1,\tfrac{2(p-i_1)}{s}-1)+\max(1,\tfrac{2(i_2-p)}{s}-1) = 1+\tfrac{2(p-i_1)}{s}-1+\tfrac{2(i_2-p)}{s}-1 = \tfrac{2(i_2-i_1)}{s}-1.\end{aligned}$$

Otherwise, we grow the outermost window pair using one of the following three cases. If there exists $p \in (j_1\,..\,i_1-s)$ such that $\mathtt{D}(j_1,j_2) = \mathtt{D}(j_1,p)+\mathtt{D}(p,j_2)$, we append $(j_1\,..\,p]$ to the window $(i_1\,..\,j_1]$. Then, the choice of p guarantees the first invariant, whereas the second invariant holds due to

$$\begin{aligned}\mathtt{D}(i_1,i_2) &= \mathtt{D}(j_1,p) + \mathtt{D}(p,j_2) + \mathsf{dyck}(x(i_1\,..\,j_1]\circ x(j_2\,..\,i_2])\\ &\ge \mathtt{D}(p,j_2) + \mathsf{dyck}(x(i_1\,..\,p]\circ x(j_2\,..\,i_2]) \ge \mathtt{D}(i_1,i_2).\end{aligned}$$

Symmetrically, if there exists $p \in (i_2-s\,..\,i_2)$ such that $\mathtt{D}(j_1,j_2) = \mathtt{D}(j_1,p)+\mathtt{D}(p,j_2)$, we prepend $(p\,..\,j_2]$ to the window $(j_2\,..\,i_2]$. By (1), the remaining case is when $\mathtt{D}(j_1,j_2) = \mathtt{D}(j_1+1,j_2-1)+\mathsf{dyck}(x[j_1+1]x[j_2])$, i.e., the optimum alignment matches $x[j_1+1]$ and $x[j_2]$. Then, we add both these characters to the outermost window pair. In this case, the first invariant holds due to $|(i_1\,..\,j_1]|,|(j_2\,..\,i_2]| < s$. As for the second invariant, we have

$$\begin{aligned}\mathtt{D}(i_1,i_2) &= \mathtt{D}(j_1+1,j_2-1) + \mathsf{dyck}(x[j_1+1]x[j_2]) + \mathsf{dyck}(x(i_1\,..\,j_1]\circ x(j_2\,..\,i_2])\\ &\ge \mathtt{D}(j_1+1,j_2-1) + \mathsf{dyck}(x(i_1\,..\,j_1+1]\circ x(j_2-1\,..\,i_2]) \ge \mathtt{D}(i_1,i_2).\end{aligned}$$

◀

Large and small windows. Let us fix an integer power of two $\theta \in [\frac{1}{n},1]$ (which will be set to $n^{-1/34}$ rounded down appropriately). For each power of two $s \in [1\,..\,\theta^{-1}]$, define a function $\mathtt{up}_s$ that maps each $i \in [0\,..\,n]$ to $\mathtt{up}_s(i) := \min(n,\theta s\lceil\frac{i}{\theta s}\rceil)$ and denote its image with N_s. Note that $N_s \subseteq [0\,..\,n]$ consists of n as well as all integer multiples of θs. Moreover, for each $i \in [0\,..\,n]$, the value $\mathtt{up}_s(i)$ is the successor of i in N_s.

We introduce the following family of variable-size windows:

$$\mathcal{I}_s := \{w \subseteq [1\,..\,n] : |w| \le s_1 \text{ and } b(w), e(w) \in N_s\}.$$

The following claim is a direct consequence of the construction.

▷ Claim 5.5. $|\mathcal{I}_s| = O(\frac{n}{\theta^2 s})$.

We pick two scales $s_1 \ge s_2$, denoting $\mathcal{J} := \mathcal{I}_{s_1}$ and $\mathcal{K} := \mathcal{I}_{s_2}$. For larger windows, we prove the following result:

▶ **Lemma 1.1.** *There exists a consistent window decomposition $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$ of $[1\,..\,n]$ such that $\sum_{(w,w')\in\mathcal{S}} \mathsf{dyck}(x[w] \circ x[w']) \leq \mathsf{dyck}(x) + 8\theta n$.*

Proof. By Lemma 5.4 applied for $s := s_1$, there exists a consistent decomposition $\mathtt{Dec}(0,n)$ of $[1\,..\,n]$ such that $\mathsf{dyck}(\mathtt{Dec}(0,n)) = \mathsf{dyck}(x)$, $|\mathtt{Dec}(0,n)| \leq \frac{2n}{s_1}$, and each window in $\mathtt{Dec}(0,n)$ is of length at most s_1. In order to meet the condition $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$, we round the window endpoints up using the $\mathtt{up}_{s_1}$ function. Formally, for each window pair $(w,w') \in \mathtt{Dec}(0,n)$, we create one window pair $(\tilde{w},\tilde{w}') \in \mathcal{S}$, where $b(\tilde{w}) = \mathtt{up}_{s_1}(b(w))$, $e(\tilde{w}) = \mathtt{up}_{s_1}(e(w))$, $b(\tilde{w}') = \mathtt{up}_{s_1}(b(w'))$, and $e(\tilde{w}') = \mathtt{up}_{s_1}(e(w'))$. The resulting family $\mathcal{S}$ satisfies $\mathcal{S} \subseteq \mathcal{J} \times \mathcal{J}$ since $|w|, |w'| \leq s_1$ implies $|\tilde{w}|, |\tilde{w}'| \leq s_1$ (because s_1 is an integer multiple of θs_1). The relative order of windows involved in $\mathcal{S}$ is the same as in $\mathtt{Dec}(0,n)$, and thus $\mathcal{S}$ remains a consistent decomposition of $(0\,..\,n]$ (this is also because $\mathtt{up}_{s_1}(0) = 0$ and $\mathtt{up}_{s_1}(n) = n$). Moreover, a single edit may increase the Dyck edit distance by at most one, and thus

$$\begin{aligned}\mathsf{dyck}(\mathcal{S}) &= \sum_{(\tilde{w},\tilde{w}')\in\mathcal{S}} \mathsf{dyck}(x[\tilde{w}] \circ x[\tilde{w}']) \leq \sum_{(w,w')\in\mathtt{Dec}(0,n)} (\mathsf{dyck}(x[w] \circ x[w']) + 4\theta s_1) \\ &= \mathsf{dyck}(\mathtt{Dec}(0,n)) + 4\theta s_1 |\mathtt{Dec}(0,n)| \leq \mathsf{dyck}(x) + 8\theta n. \end{aligned}$$ ◀

Our next objective is to estimate $\mathsf{dyck}(x[w] \circ x[w'])$ for each $w, w' \in \mathcal{J}$. For this, we utilize the smaller windows via the following result. Its proof, similar to that of Lemma 1.1, is left for the full version [18].

▶ **Lemma 5.6.** *For every $(w,w') \in \mathcal{J} \times \mathcal{J}$ with $e(w) \leq b(w')$, there exists a consistent window decomposition $\mathcal{S} \subseteq \mathcal{K} \times \mathcal{K}$ of $w \cup w'$ such that $\mathsf{dyck}(\mathcal{S}) \leq \mathsf{dyck}(x[w] \circ x[w']) + O(\theta|w \cup w'|)$.*

5.2 Outline of the Proof of Theorem 5.2

We set θ, s_1, s_2 to be the largest integer powers of two satisfying $s_1 \leq n^{21/34}$, $s_2 \leq n^{13/34}$, and $\theta \leq n^{-1/34}$, respectively. We first construct the families $\mathcal{J}$ and $\mathcal{K}$ of large and small windows, as defined in Section 5.1. Then, we run a procedure `CertifyWindowPairs` (described in the full version [18]), which certifies window pairs in $\mathcal{J} \times \mathcal{J}$ and $\mathcal{K} \times \mathcal{K}$; some pairs are certified directly, using Theorem 3.2 with $\epsilon = 1$ (which provides a 2-approximation), whereas others indirectly, using the triangle inequality (Lemma 2.3). The resulting family $\mathcal{W}$ of certified window pairs satisfies the following property with probability $1 - n^{-9}$: there exists a consistent decomposition $T \subseteq \mathcal{W}$ of $[1\,..\,n]$ such that $\sum_{(w,w',c)\in T} c \leq 40 \cdot \mathsf{dyck}(x) + O(\theta n)$.

Next, we use a simple dynamic-programming procedure (described in the full version [18]) to minimize the total cost $\sum_{(w,w',c)\in\tilde{T}} c$ among all consistent decompositions $\tilde{T} \subseteq \mathcal{W}$ of $[1\,..\,n]$. The resulting cost is at least $\mathsf{dyck}(x)$ because $\mathcal{W}$ contains certified window pairs only (that is, $c \geq \mathsf{dyck}(x[w] \circ x[w'])$ holds for each $(w,w',c) \in \mathcal{W}$). Moreover, the cost is at most $40 \cdot \mathsf{dyck}(x) + O(\theta n) \leq 40 \cdot \mathsf{dyck}(x) + O(n^{33/34})$ by the existence of T. The running time of the DP procedure is $\tilde{O}(|N_{s_2}|^3 + |\mathcal{W}|)$, which is $\tilde{O}(n^{67/34})$ by the choice of parameters. The details of the running-time analysis are left for the full version [18].

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. *SIAM Journal on Computing*, 47(6):2527–2555, 2018. doi:10.1137/16M1061771.

2 Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4), 1972. doi:10.1137/0201022.

3 Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001. doi:10.1137/s0097539700366528.

4 Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *FOCS 2010*, pages 377–386. IEEE, 2010. doi:10.1109/focs.2010.43.

5 Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In *FOCS 2020*, pages 990–1001. IEEE, 2020. doi:10.1109/focs46700.2020.00096.

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

7 Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In *PODS 2016*, pages 477–488. ACM, 2016. doi:10.1145/2902251.2902304.

8 Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. In *FOCS 2004*, pages 550–559. IEEE, 2004. doi:10.1109/FOCS.2004.14.

9 Tugkan Batu, Funda Ergün, and Süleyman Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *SODA 2006*, pages 792–801. ACM, 2006. doi:10.1145/1109557.1109644.

10 Djamal Belazzougui and Qin Zhang. Edit distance: Sketching, streaming, and document exchange. In *FOCS 2016*, pages 51–60. IEEE, 2016. doi:10.1109/FOCS.2016.15.

11 Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. *Journal of the ACM*, 68(3):19:1–19:41, 2021. doi:10.1145/3456807.

12 Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: Beyond worst case. In *SODA 2020*, pages 1601–1620. SIAM, 2020. doi:10.1137/1.9781611975994.99.

13 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 Journal on Computing*, 48(2):481–512, 2019. doi:10.1137/17M112720X.

14 Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM Journal on Computing*, 42(1):61–83, 2013. doi:10.1137/100816481.

15 Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. *Journal of the ACM*, 67(6):1–22, 2020. doi:10.1145/3422823.

16 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *STOC 2016*, pages 712–725. ACM, 2016. doi:10.1145/2897518.2897577.

17 Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. Faster min-plus product for monotone instances. In *STOC 2022*. ACM, 2022. arXiv:2204.04500, doi:10.48550/arXiv.2204.04500.

18 Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for dyck edit distance and RNA folding, 2021. arXiv:2112.05866.

19 Eldar Fischer, Frédéric Magniez, and Tatiana Starikovskaya. Improved bounds for testing Dyck languages. In *SODA 2018*, pages 1529–1544. SIAM, 2018. doi:10.1137/1.9781611975031.100.

20 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

21 Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. An improved algorithm for the k-Dyck edit distance problem. In *SODA 2022*. SIAM, 2022.

22 Elazar Goldenberg, Aviad Rubinstein, and Barna Saha. Does preprocessing help in fast sequence comparisons? In *STOC 2020*, pages 657–670. ACM, 2020. doi:10.1145/3357713.3384300.

23 Michael A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

24 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: https://depotuw.ceon.pl/handle/item/3614.

25 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

26 Dexter C. Kozen. *Automata and Computability*. Springer New York, 1997. doi:10.1007/978-1-4612-1844-9.

27 Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *MFCS 2011*, volume 6907 of *LNCS*, pages 412–423. Springer, 2011. doi:10.1007/978-3-642-22993-0_38.

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

29 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

30 Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014. doi:10.1137/130926122.

31 Ruth Nussinov and Ann B Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences*, 77(11):6309–6313, 1980. doi:10.1073/pnas.77.11.6309.

32 Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Structures & Algorithms*, 22(1), January 2003. doi:10.1002/rsa.10067.

33 Aviad Rubinstein, Saeed Seddighin, Zhao Song, and Xiaorui Sun. Approximation algorithms for LCS and LIS with truly improved running times. In *FOCS 2019*, pages 1121–1145. IEEE, 2019. doi:10.1109/focs.2019.00071.

34 Barna Saha. The Dyck language edit distance problem in near-linear time. In *FOCS 2014*, pages 611–620. IEEE, 2014. doi:10.1109/focs.2014.71.

35 Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *FOCS 2015*, pages 118–135. IEEE, 2015. doi:10.1109/focs.2015.17.

36 Barna Saha. Fast & space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In *FOCS 2017*, pages 295–306. IEEE, 2017. doi:10.1109/FOCS.2017.35.

37 Masoud Seddighin and Saeed Seddighin. $3+\epsilon$ approximation of tree edit distance in truly subquadratic time. In *ITCS 2022*, volume 215, pages 115:1–115:22, 2022. doi:10.4230/LIPIcs.ITCS.2022.115.