Fast Algorithms for Parsing Sequences of Parentheses with Few Errors

Arturs Backurs
MIT

Krzysztof Onak
IBM Research

Abstract

We consider the problem of fixing sequences of unbalanced parentheses. A classic algorithm based on dynamic programming computes the optimum sequence of edits required to solve the problem in cubic time. We show the first algorithm that runs in linear time when the number of necessary edits is small. More precisely, our algorithm runs in $O(n) + d^{O(1)}$ time, where n is the length of the sequence to be fixed and d is the minimum number of edits.

The problem of fixing parentheses sequences is related to the task of repairing semi-structured documents such as XML and JSON.

1 Introduction

Balanced sequences of parentheses can be used to describe arbitrary rooted trees. Their popular application is to specify the order in which binary operations are performed in algebraic and arithmetic expressions. The language of well-balanced sequences on k different types of parentheses is also known as the Dyck(k) language and can easily be expressed as a context-free grammar.

The Dyck(k) language is inherently present in both numerous semi-structured document formats—such as XML and JSON—and essentially all programming languages that allow for recursively nesting blocks and loops. In XML and—most of the time—in HTML,[1] an opening tag `<xyz>` has to be accompanied by the corresponding closing tag `</xyz>`, where `xyz` is an arbitrary allowed identifier. Pairs of tags can be nested as in `<a><b></b></a>`. They can also appear alongside as in `<a></a><b></b>`, but they cannot be interleaved as in `<b><a></b></a>`.

A large fraction of HTML documents on the web are malformed or do not conform to HTML standards. Unfortunately, there is no mechanism allowing to notify the content creator about the issue and speedily receive a corrected document. Therefore, at the risk of losing their market share, major web browsers are forced to routinely display malformed HTML documents. While many tags such as section headers cannot be meaningfully nested, an arbitrary proper nesting of text formatting elements such as `<i>`, `<b>`, `<sub>`, `<sup>`, etc. is allowed within a paragraph.[2]

Given an incorrectly formatted text paragraph, what is the best way to interpret it in order to visualize it? One way to approach this problem is to find a way of fixing the formatting by making the minimum number of tag modifications such as substitutions, deletions, and insertions. This corresponds to expressing the formatting tags as a sequence of parentheses and finding the minimum number of edits converting the input sequence into a sequence in Dyck(k).

Unfortunately, the best known algorithm for this problem requires time that is cubic in the length of the input (it also uses quadratic space), which is prohibitively expensive even for relatively small documents. A linear time algorithm would be ideal from the practical perspective, but there is evidence that such algorithms may not exist [ABVW15]. However, this does not preclude the existence of more efficient algorithms if the edit distance is *small*. If the distance to Dyck(k) is large, then the document is likely to be too corrupt to permit a meaningful recovery.

In this paper we focus on the small distance regime, and show an algorithm that finds a minimum number of edits in time and space $O(n) + d^{O(1)}$, where n is the length of the sequence and d is the distance to Dyck(k).

Potential applications of our algorithm go beyond correcting HTML documents. Our techniques can be applied to correcting a wider class of semi-structured documents, the topic discussed in the papers by Korn et al. [KSSY13] and Saha [Sah14]. The work by Korn at al. [KSSY13] gives an algorithm that returns few edits needed to repair a sequence of parentheses. It discusses the practical aspects of parentheses repairing in depth. Furthermore, compilers often do not stop processing the input immediately after encountering the first syntax error but attempt to correct it in order to list more errors at once. Frequently, syntax errors are a result of incorrectly nested blocks. The authors of this paper have suffered from mismatched LaTeX `\begin{...}` and `\end{...}` tags multiple times while writing this work. A fast tag correcting algorithm could also be used in an integrated development environment to provide feedback to the user about structural problems in the document being created. The Dyck language arises naturally in many biological applications like RNA folding [VGF14].

[1] Formally, HTML is not an XML-adherent language. Attempts at making it follow a more restrictive set of rules, such as XHTML, have not been very widely adopted. However, the differences are not important for the discussion here, which focuses on an XML-like part of HTML.

[2] See any HTML standard definition, e.g., [htm], or tutorial for more information on the function of these tags.

1.1 Our results

Table 1 summarizes previous results and our results on computing distance to Dyck(k). We give two algorithms. We start our presentation with an easier and faster algorithm for the case that only deletions are allowed. Its running time is $O(n+d^6)$, where n is the length of the sequence and d is the distance. If substitutions are allowed as well, we give a slower $O(n+d^{16})$-time algorithm. In either case, the running time does not depend on k as long as all parentheses are represented as positive integers bounded by $n^{O(1)}$. (Under this condition, an underlying suffix tree can be built in $O(n)$ time.)

	Runtime	Approximation factor	Comments
Theorem 40	$O(n+d^{16})$	Exact	
Theorem 26	$O(n+d^6)$	Exact	No substitutions
[AP72, Mye95]	$O(n^3)$	Exact	
	$O\left(n^3/2^{\Omega(\sqrt{\log n})}\right)$	Exact	Follows from [Wil14] and [Val75]
	$O(d \cdot n^\omega)$	Exact	Follows from [GM97, SZ99] and [Val75]
[Sah14]	$O(n^{1+\epsilon})$	$(\log n)^{O(1/\epsilon)}$	
[Sah14]	$O(n)$	$O(d)$	
[Sah14]	$O(n+d^2)$	$O(\log d)$	
	$2^{O(d)} \cdot n$	Exact	Follows from [Sah14]
[Sah15]	$\tilde{O}\left(n^\omega/\epsilon^{O(1)}\right)$	$1+\epsilon$	
[ABVW15]	$\Omega\left(n^{\omega-o(1)}\right)$	Exact	Conditional lower bound

Table 1: Summary of our and previous results on computing distance to Dyck(k). n is the length of the sequence of parentheses. d is the upper bound on the Parentheses language edit distance. $\omega < 2.373$ is the matrix multiplication constant [Wil12, Gal14].

A note on computing an optimal sequence of edits For simplicity, we focus our presentation on computing distance to the closest well-balanced sequence of parentheses. In a more practical setting, one would most likely be interested in obtaining an optimal sequence of edits as well. The sequence of edits provides a possible way of interpreting the corrupt input sequence as a well balanced sequence of parentheses. Our algorithms construct and retain a dynamic programming table that is sufficient for reconstructing the optimal sequence of edits in the same running time. We note that if the distance is larger than d, our algorithms detect this fact in the same running time.

A note on the presentation In the presentation of our algorithms, we assume that d is an upper bound on edit distance to Dyck(k). Our general result can be obtained by first setting $d = 1$ and then doubling d as long as the algorithm fails. This is enough to obtain an $O(n \log d) + d^{O(1)}$-algorithm, where d is the actual distance. The part of our algorithm that takes linear time is preprocessing, which is independent of the bound on d. Hence the $O(\log d)$ factor in the running time is not necessary.

1.2 Previous results and related work

We list related results in Table 1. A cubic-time algorithm for computing distance to Dyck(k) was given by Aho and Peterson [AP72], who studied the distance to general context-free grammars (see also [Mye95]). Valiant [Val75] studied subcubic algorithms for the problem. Combining his techniques

with state-of-the-art algorithms for distances in graphs, one obtains exact algorithms of running times $O(n^3/2^{\Omega(\sqrt{\log n})})$ and $O(d \cdot n^\omega)$.

Saha [Sah14] studied the problem of *approximating* the distance to Dyck(k) in near-linear time. She gave meta-algorithms that reduce the problem to a small number of executions of an approximate edit distance algorithm. In Table 1, we list three applications of her results. The most related to the topic of this paper is an algorithm that computes an $O(\log d)$-approximation in $O(n + d^2)$ time. Her meta-algorithms are based on cleverly guessing an alignment of strings with a neat analysis based on random walks. Since whenever a parsing algorithm gets stuck, the optimal edit decision that enables proceeding comes from a constant-size set of options, one can enumerate over all possible sets of at most d edit decisions in order to find an optimal edit sequence. This leads to an exact algorithm that runs in $2^{O(d)} \cdot n$ time. In another work, Saha [Sah15] gave an algorithm for computing a $(1+\epsilon)$-approximation to distance to a fixed context-free grammar. The running time of the algorithm is $\tilde{O}(n^\omega/\epsilon^{O(1)})$.

Finally, Abboud et al. [ABVW15] showed that under plausible complexity assumptions, there is no exact algorithm that runs significantly faster than in time $O(n^\omega)$. Hence, it is unlikely that there is an exact algorithm for distance to Dyck(k) that runs in near-linear time even for large d.

1.3 Our techniques

We first preprocess the input sequence to align pairs of parentheses that can already be matched. Eliminating them does not change the distance to Dyck(k). After this step, one can show that the input consists of $O(d)$ blocks, where each block is first a sequence of opening and then closing parentheses. We also define a height function that describes how deeply nested each parenthesis is. Parentheses at very different heights can only be matched with a large number of edits. This together with a bound on the number of blocks leads to a poly(d) bound on what other parentheses each parenthesis could be matched to (perhaps after a substitution).

Our algorithms enumerate over all possible matches of borders between blocks in the above decomposition. The number of all such matches can be shown to be polynomial in d. Assuming that we know how to efficiently compute the number of edits required to align potentially long but very simple sequences of opening—on one side—and closing—on the other side—parentheses, we build a dynamic programming instance on top of poly(d) different subsequences. The dynamic program gives eventually the solution to the edit distance problem.

The missing piece, a subroutine for quickly computing how well two "simple" subsequences of parentheses align, is an adaptation of techniques from the $O(n + d^2)$-time edit distance algorithm [LMS98], which largely inspired our work. A "simple" subsequence consists of only opening or closing parentheses. In particular, out algorithms have have to align very distant subsequences of parentheses efficiently. This is what our adaptation of techniques of Landau et al. [LMS98] can do in $O(d^2)$ time, independently of the length of subsequences involved, after an initial linear-time preprocessing.

1.4 Future work

We see our algorithm as a proof of concept that near-linear time algorithms can be constructed when the edit distance to Dyck(k) is polynomially small. We believe that further research will result in improved running time and, simultaneously, more practical algorithms. We predict, however, that constructing an $O(n + d^3)$-time algorithm will turn out to be a very challenging task. If it is not possible at all, it would be very interesting to prove a conditional lower bound that uses a popular complexity assumption that computing edit distance to Dyck(k) requires $\omega(n)$ time for $d = o(n^{1/3})$ (see [BI15, ABVW15] for examples of related conditional lower bounds).

In the long run, it would be interesting to develop similar algorithms for a wider class of grammars relevant for programming languages.

2 Preliminaries

We write $[n]$ to denote the set $\{1, 2, \ldots, n\}$ for any integer n. For integers n, m, we write $[n, m]$ to denote $\{n, \ldots, m\}$. For a sequence X, $|X|$ denotes its length (the number of symbols in it) and X_i denotes the i-th symbol of X for $i \in [|X|]$. For a sequence X of symbols, $\mathrm{rev}(X)$ is X reversed. More precisely, for every $i \in [|X|]$, we set $(\mathrm{rev}(X))_i := X_{|X|+1-i}$. Given two sequences X and Y, we write $X \circ Y$ to denote the product of their concatenation. From now on, whenever we use the variable k, it will *not* be the number of different types of parentheses k (as in $\mathrm{Dyck}(k)$). Remember that our runtimes do not depend on k (as long as integers representing parentheses are reasonably bounded).

Parentheses The input sequences consist of parentheses of different types. For two matching parentheses, one of them is *opening* and the other is *closing*. They can match only if the opening occurs before the closing. Under the usual interpretation, "`(`" and "`)`" are a matching pair of parentheses, and so are "`{`" and "`}`." Among them, "`(`" and "`{`" are opening parentheses, and "`)`" and "`)`" are closing parentheses.

Definition 1. *We say that two parentheses are of the same* type *if they are equal or can be matched.*

According to the above definition, "`(`" and "`)`" are of the same type, "`(`" and "`(`" are also of the same type, but "`(`" and "`]`" are *not* of the same type.

We define an operation that removes a sequence of parentheses the information about whether each parenthesis is opening or closing. It keeps, however, the information about the type of each of them.

Definition 2. *We assign integers* 1, 2, ... *to all possible types of parentheses. Given a sequence X of parentheses, we define a sequence $U(X)$ of symbols as follows. For every $i \in [|X|]$, if the parenthesis X_i belongs to the j-th type (it can be a closing or an opening parenthesis), we replace it with a formal symbol a_j.*

For completeness, we state a recursive definition of balanced sequences.

Definition 3. *A sequence X of parentheses is balanced if any of the following holds:*

- *X is an empty string,*
- *$X = Y \circ Z$, where Y and Z are non-empty balanced sequences,*
- *$X = Y_1 \circ Z \circ Y_2$, where Z is a balanced sequence and Y_1 are Y_2 are sequences of length 1 that contain matching opening and closing parentheses, respectively.*

As an example, "`(){}`" is balanced and "`(()()`" is not.

Distance We consider two distance measures. In one, only deletions are allowed. In the other, substitutions are allowed as well.

Definition 4. *Let S be a sequence of possibly unbalanced parentheses. We write*

- $\mathrm{edit}_1(S)$ *to denote the minimum number of parenthesis deletions necessary to turn S into a balanced sequence of parentheses,*
- $\mathrm{edit}_2(S)$ *to denote the minimum number of parenthesis deletions and substitutions (of one parenthesis with another) necessary to turn S into a balanced sequence of parentheses.*

3 Algorithm for distance with no substitutions

We start by describing a simpler and faster algorithm for the case of only deletions allowed. When substitutions are allowed, we have to handle a few more cases, which inflate both the descriptive and computational complexity of the algorithm.

3.1 Fast distance computation for a simple case

In this subsection, we describe a subroutine that we repeatedly use for computing edit distance (with no substitutions) when the sequence consists of first opening and then closing parenthesis. More generally, we construct a data structure that allows for efficiently computing this kind of information, when the two parts of the sequence are arbitrary disjoint subsequences of the input. We heavily borrow techniques from Landau et al. [LMS98], who designed them for their $O(n + d^2)$ edit distance algorithm.

We start with a definition.

Definition 5. *Let X be a sequence of opening parentheses and Y be a sequence of closing parentheses. By* $\mathrm{edit}_1(X, Y)$ *we denote the quantity* $\mathrm{edit}_1(XY)$.

There is a simple algorithm based on dynamic programming for computing $\mathrm{edit}_1(X, Y)$ in time $O(|X| \cdot |Y|)$. We describe the algorithm here in full because this allows us to introduce a notation that we will find helpful later.

Definition 6. *For any sequences A and B, we define* $\mathrm{edit}'_1(A, B)$ *as the minimum total number of symbol deletions in A and B necessary to make both sequences equal.*

The following fact is straightforward (see Definition 2 for U).

Fact 7. $\mathrm{edit}_1(X, Y) = \mathrm{edit}'_1(U(X), \mathrm{rev}(U(Y)))$.

Therefore, to compute $\mathrm{edit}_1(X, Y)$, it suffices to compute $\mathrm{edit}'_1(U(X), \mathrm{rev}(U(Y)))$. We now show that, given two sequences A, B of symbols, $\mathrm{edit}'_1(A, B)$ can be computed in $O(|A| \cdot |B|)$ time.

The algorithm for $\mathrm{edit}'_1(A, B)$ proceeds as follows. It builds dynamic programming table D with entries $D_{r,c}$ for $r = 0, \ldots, |A|$ and $c = 0, \ldots, |B|$. It sets $D_{0,c} = c$ for all c and $D_{r,0} = r$ for all r. Then the algorithm fills in the rest of the table D using the following rule:

$$D_{r,c} := \min(D_{r-1,c} + 1, D_{r,c-1} + 1, 2 \cdot \delta_{A_r \neq B_c} + D_{r-1,c-1}), \tag{1}$$

where $\delta_E = 1$ if event E happens and $\delta_E = 0$ if event E does not happen. The last entry $D_{|A|,|B|}$ that we fill in equals $\mathrm{edit}'_1(A, B)$.

Definition 8. *For any integer i, let $T_i := \{(r, c) \mid r - c = i\}$ be the i-th diagonal of entries of D.*

The following two properties of D follow from (1).

Property 9. *For any r and c, $D_{r+1,c+1} \geq D_{r,c}$.*

Property 9 says that if we move along any diagonal T_i in D, the entries do not decrease.

Property 10. *Let d be an arbitrary integer. For any integer i with $|i| > d$, we have that $D_{r,c} > d$ for any $(r, c) \in T_i$.*

Property 10 says that all entries in a diagonal far from the main diagonal are large. This implies that if we know that the overall distance is at most d, we do not need to compute entries in diagonals T_i for $|i| > d$ in order to compute the total distance between the strings.

Definition 11. *For integer $i = 0, \ldots, d$, we define*

$$\text{wave}(i) := \{(r, c) \mid D_{r,c} = i \text{ and } D_{r+1,c+1} > i\}.$$

Theorem 12. *Let C be a sequence of symbols of length n. We can preprocess C in time $O(n)$ so that the following holds. Let $d \geq 0$ be an arbitrary integer, independent of the preprocessing step. Given two arbitrary substrings C_1 and C_2 of C, we can output* $\text{wave}(0), \ldots, \text{wave}(d)$ *(corresponding to* $\text{edit}'_1(C_1, C_2)$*) in time $O(d^2)$.*

Proof. By Property 10, we have that $|\text{wave}(i)| \leq 2d + 1$ for all $i = 0, \ldots, d$. This is because $\text{wave}(i)$ contains at most one (r, c) from each diagonal T_j. The total size of the waves is at most $O(d^2)$. The proof of the theorem appears in [LMS98] (see subsection 2.3 in [LMS98]). The theorem is proved for a slightly different matrix D. It is proved for the case when $\text{edit}'_1(C_1, C_2)$ is the minimum number of insertions, deletions, *and substitutions* necessary to transform C_1 into C_2. However, the proof can be straightforwardly generalized to our definition of edit'_1. More precisely, we change Lemma 2.8 from [LMS98] by replacing expression $\max\{L^{h-1}(d+1)+1, L^{h-1}(d)+1, L^{h-1}(d-1)+1\}$ with $\max\{L^{h-1}(d+1)+1, L^{h-1}(d-1)+1\}$, that is, we remove the second argument from max. Also, notice that we added "+1" to the third argument. We believe that the current statement of Lemma 2.3 in [LMS98] without "+1" is not correct.

The only application of the suffix tree both in [LMS98] and in this theorem is for finding the length of the longest identical substring starting at two specified indices in constant time. Formally, given a sequence $S = S_1 \ldots S_n$, we want to answer queries of the following form: given i and j, what is the largest q such that for all $t \in \{0, \ldots, q-1\}$, $S(i+t) = S(j+t)$? The linear time precomputation involves building the suffix tree and constructing an LCA (=lowest common ancestor) data structure on the suffix tree. The answer to queries can be provided in constant time by finding the leaves corresponding to the suffixes starting at i and j and finding their LCA. The weighted depth of the LCA provides the length of the string.

The algorithm described in Section 2.3 of [LMS98] computes consecutive waves $\text{wave}(0), \ldots, \text{wave}(d)$ using only answers to the above queries, where $S = AxYy$ and x and y are unique symbols (x and y can be removed if one trims the answers provided by the data structure). The algorithm doesn't need to know anything else about the input. Therefore, the same data structure works for arbitrary substrings C_1 and C_2 of any fixed string $S = C$ (as long as we trim the answers to within C_1 and C_2). The precomputation has to be done only once for the entire string S. This is how our algorithm works. For any pair of substrings, the computation time and number of queries to the data structure are bounded by $O(d^2)$. □

Theorem 13. *Let C be a sequence of symbols of length n. We can preprocess C in time $O(n)$ so that the following holds. Let $d \geq 0$ be an arbitrary integer, independent of the preprocessing step. Given two arbitrary substrings C_1 and C_2 of C, we can output a data structure in time $O(d^2)$ that satisfies the following property. Let D be the dynamic programming table corresponding to* $\text{edit}'_1(C_1, C_2)$*. Given arbitrary (r, c), we can in $O(\log d)$ time output $D_{r,c}$, if $D_{r,c} \leq d$. If $D_{r,c} > d$, we output that $D_{r,c} > d$ in constant time by comparing (r, c) to* $\text{wave}(d)$*.*

Proof. By Theorem 12, given two substring C_1, C_2, we can in $O(d^2)$ output $\text{wave}(0), \ldots, \text{wave}(d)$. This is our data structure. Suppose we are given arbitrary (r, c). If $D_{r,c} \leq d$, we can output $D_{r,c}$ in $O(\log d)$ by doing binary search among values $\text{wave}(i)$. We can do binary search because $\text{wave}(0) \cap T_{r-c}, \ldots, \text{wave}(d) \cap T_{r-c}$ are nondecreasing by Property 9. We store each $\text{wave}(i)$ indexed by diagonals $-d, \ldots, -1, 0, 1, \ldots, d$. If $D_{r,c} > d$, we output that $D_{r,c} > d$ in constant time. □

We use the following theorem later in the paper.

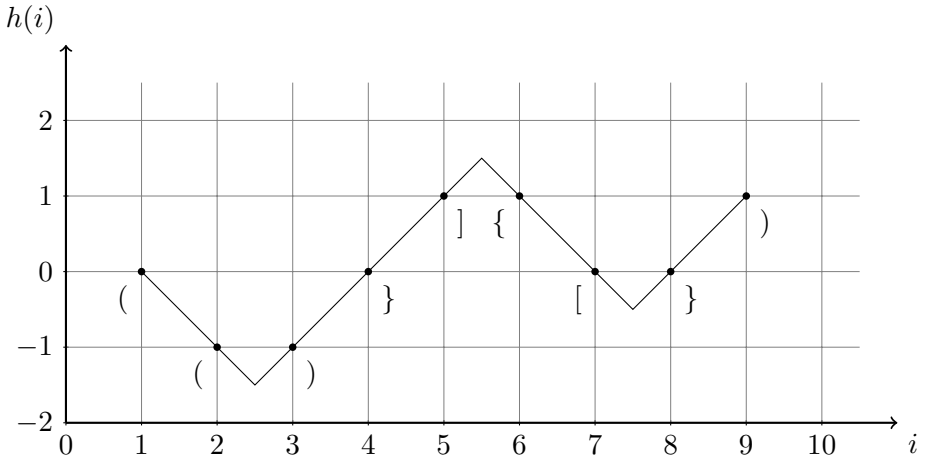

Figure 1: Height function $h : [9] \to \mathbb{Z}$ corresponding to sequence $S = (()\}]\{[\})$. See Definition 15.

Theorem 14. *For any sequence S of parentheses, we can preprocess S in time $O(n)$, where $n = |S|$, so that the following holds. Given any two substrings S_1 and S_2 of S and a parameter d, we can, in $O(d^2)$ time, build a data structure, corresponding to S_1 and S_2, that provides the following two operations:*

- *For any prefix S_1' of S_1 and any suffix S_2' of S_2, such that $\mathrm{edit}_1(S_1', S_2') \leq d$, we can output $\mathrm{edit}_1(S_1', S_2')$ in $O(\log d)$ time.*
- *We can check whether $\mathrm{edit}_1(S_1', S_2') \leq d$ in $O(1)$ time.*

Proof. We set $C := U(S) \circ \mathrm{rev}(U(S))$ and preprocess C as in Theorem 13. $U(S_1)$ is substring of $U(S)$. $\mathrm{rev}(U(S_2))$ is substring of $\mathrm{rev}(U(S))$. $U(S_1')$ is a prefix of $U(S_1)$. $\mathrm{rev}(U(S_2'))$ is a prefix of $\mathrm{rev}(U(S_2))$. By Fact 7 and Theorem 13, we get the required algorithm. □

3.2 Algorithm

Let S be a sequence of parentheses. We denote the length of S by $n = |S|$. We are given integer d such that $\mathrm{edit}_1(S) \leq d$. In this section we show how to compute $\mathrm{edit}_1(S)$ in time $O(n + d^6)$.

The following height function is very convenient in further analysis.

Definition 15. *We define function $h : [n] \to \mathbb{Z}$ as follows: $h(1) := 0$ and, for $n \geq i \geq 2$,*

$$h(i) := \begin{cases} h(i-1) & \text{if } S_{i-1} \text{ and } S_i \\ & \text{are of different classes;} \\ h(i-1) - 1 & \text{if } S_{i-1} \text{ and } S_i \text{ are both opening;} \\ h(i-1) + 1 & \text{if } S_{i-1} \text{ and } S_i \text{ are both closing.} \end{cases}$$

We see that function h decreases if it has consecutive opening parentheses and increases if it has consecutive closing parentheses. Otherwise function does not change its value. See Figure 1 for an example and visualization. Consider the i-th symbol of S, we call $h(i)$ the height of the i-th symbol. h is the height function for sequence S.

We decompose sequence S as

$$S = D_1 U_1 D_2 U_2 \dots D_{k-1} U_{k-1} D_k U_k \tag{2}$$

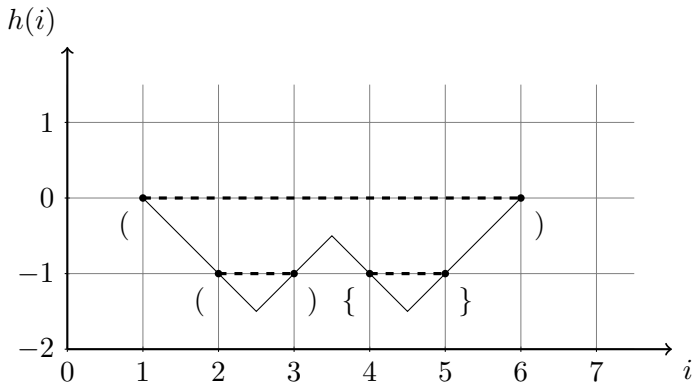

Figure 2: Height function $h : [6] \to \mathbb{Z}$ corresponding to a balanced sequence of parentheses $S = (()\{\})$. Dashed lines connect the aligned pairs of sequences. Notice that the horizontal dashed lines do not cross the graph of the function h.

for some integer $k \geq 1$. Each D_i consists of opening parentheses only and each U_i consists of closing parentheses only. We require that all U_i, $i < k$ are non-empty sequences. We require that all D_i, $i > 1$ are non-empty sequences. Only D_1 and U_k may be empty. Note that the decomposition is unique.

Definition 16. *For $i \in [k]$, we call sequence D_iU_i the i-th* valley *of S.*

Definition 17. *If D_1 is nonempty, we call the first symbol of D_1 the 0-th peak of S. We call the last symbol of U_1 and the first symbol of D_2 the 1-st peak of S. In general, for $i = 1, \ldots, k-1$, we call the last symbol of U_i and the first symbol of D_{i+1} the i-th peak of S. If U_k is nonempty, we call the last symbol of U_k the k-th peak of S.*

Consider any series of deletions of parentheses that produce a balanced final sequence S' of parentheses. If we visualize the height function of S', we observe the following property. Consider any two parentheses that are aligned (in particular, they are of the same type, the left one is an opening parenthesis and the right one is a closing parenthesis). These have the same height and, if we connect them with a horizontal line, there will be no intersection with the graph of the height function. See an example in Figure 2.

Consider the aligned parentheses in the original sequence S. It is possible to connect all pairs of the aligned parentheses without one line crossing another line or a line crossing the graph of the height function. See Figure 3 for an example. We call the set of aligned parentheses an alignment of S.

Now we describe a recursive algorithm that computes edit_1 distance. Its running time is large, but after a series of improvements, we obtain the desired running time of $O(n + d^6)$.

We first preprocess S as follows. As long as there are two *neighboring* symbols that can be aligned, remove them. Let S'' be the resulting sequence after the preprocessing. We state the next simple fact without a proof.

Fact 18. $\text{edit}_1(S) = \text{edit}_1(S'')$.

The intuition is that if we don't match the two symbols (one of them is matched to a symbol somewhere else), one of the two symbols will be deleted or substituted which contributes 1 to the distance. We can rematch the two symbols in a pair and we don't increase the total distance.

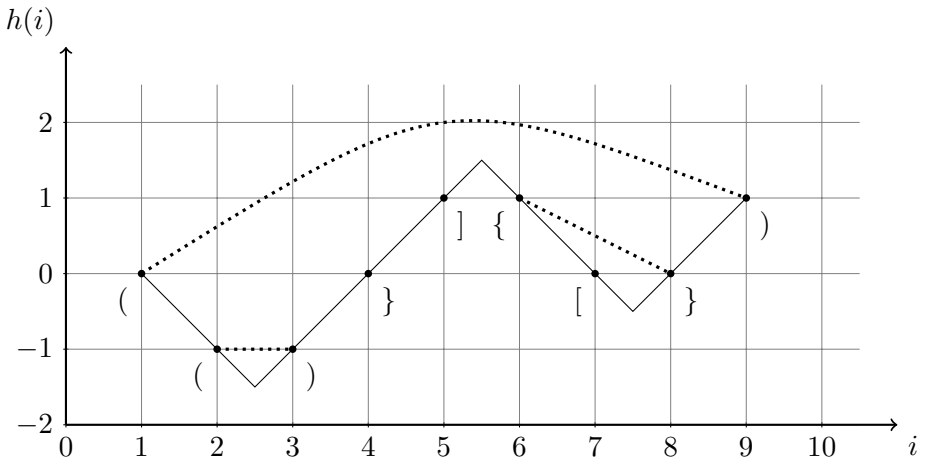

Figure 3: Height function $h : [9] \to \mathbb{Z}$ corresponding to an unbalanced sequence of parentheses $S = (()\}]\{[\})$. Notice that the balanced sequence of parentheses $S' = (()\{\})$ corresponding to Figure 2 is subsequence of S. For every aligned pair of parentheses in S', we connect the corresponding parentheses with dotted line in the graph corresponding to sequence S. Notice that the dotted lines do not cross the graph of the function h. The dotted lines can't be always drawn horizontally as opposed to the dashed lines in Figure 2.

We can check that the preprocessing can be done in time $O(n)$. Therefore, because of this and Fact 18, we can assume that the input sequence is already preprocessed. Thus, we can assume that S satisfies the following property.

Property 19. *No two neighbouring symbols of S can be aligned.*

We use several times the following simple fact which we state without a proof.

Fact 20. *Consider subsequence $S_iS_{i+1}\dots S_j$ of S for some integers $j \geq i$. If $|h(j) - h(i)| > d$, then*

$$\mathrm{edit}_1(S_iS_{i+1}\dots S_j) > d.$$

Claim 21. *$k \leq d$ (see (2) for the definition of k).*

Proof. Suppose that D_1 and U_k are not empty. By Property 19, the last symbol of D_i and the first symbol of U_i cannot be aligned for every $i \in [k]$. It means that at least one of them has to be deleted. Therefore, $\mathrm{edit}_1(S) \geq k$ and the claim follows from the promise that d upper bounds $\mathrm{edit}_1(S)$. If D_1 is empty then the first symbol of U_1 must be deleted. If U_k is empty the the last symbol of D_k must be deleted. If one of these cases or both cases happen, we still have that $\mathrm{edit}_1(S) \geq k$. Which implies that $k \leq d$. □

3.2.1 First improvement

In this section we prove the following theorem.

Theorem 22. *Let S be a sequence of parentheses that satisfy Property 19. Assume that* $\mathrm{edit}_1(S) \leq d$. *Then we can compute* $\mathrm{edit}_1(S)$ *in time $O(n^2d^6)$.*

Notice that this running time is already better by a polynomial factor than the trivial $O(n^3)$ running time when $d \leq o(n^{\frac{1}{6}})$.

The recursive algorithm is as follows. It considers three cases.

Case 1 $k = 1$. In this case we compute $\mathrm{edit}_1(D_1, U_1)$ and output the result.

If Case 1 does not occur (that is, $k > 1$), we consider the next two cases. Case 2 results in quantity M_1, Case 3 results in quantity M_2. The algorithm outputs $\min(M_1, M_2)$.

Case 2 There is an optimal alignment for S in which there is a parenthesis in D_1 that is aligned to a parenthesis in U_k. We call such aligned pair of parentheses *good*. Fix an optimal alignment that has a good pair. Choose good pair for which the parenthesis in D_1 is the rightmost one in D_1 among all good pairs. Suppose that this good pair aligns parenthesis S_i (in D_1) with parenthesis S_j (in U_k). We write $D_1 = D_1' D_1''$ such that D_1' ends with S_i. Similarly, we write $U_k = U_k'' U_k'$ such that U_k' starts with S_j.

Lemma 23. *There exists $r \in \{1, \ldots, k-1\}$ with the following property. There is no symbol in any of the sequences*

$$D_1'', D_2, D_3, D_4, \ldots, D_r$$

that is matched to a symbol from any of the sequences

$$U_{r+1}, \ldots, U_{k-3}, U_{k-2}, U_{k-1}, U_k''.$$

Proof. Consider $r = k - 1$. If this r does not satisfy the condition, there must be alignment of U_k'' with $D_{r'}$ for $r \geq r' \geq 2$ or with D_1''. We can exclude alignment between U_k'' and D_1'' because that contradicts the choice of S_i and S_j. Consider $r = r' - 1$. If this r does not satisfy the condition, there must be alignment of U_k'' with $D_{r''}$ for $r \geq r'' \geq 2$. Consider $r = r'' - 1$. We continue this until $r = 1$ or we find r that satisfies the condition in the statement of the lemma. If we continue until $r = 1$, this value $r = 1$ satisfies the condition because otherwise there is an alignment between D_1'' and U_k'', which contradicts the choice of S_i and S_j. □

By the assumption, there is an optimal alignment for S in which there is a parenthesis in D_1 that is aligned to a parenthesis in U_k. This means that we can write $D_1 = D_1' D_1''$ and $U_k = U_k'' U_k'$ as described above. Now, using Lemma 23, we can split the problem into three independent parts for some $r \in \{1, \ldots, k-1\}$.

$$\begin{aligned}\mathrm{edit}_1(S) = \ & \mathrm{edit}_1(D_1', U_k') \\ & + \mathrm{edit}_1(D_1'' U_1 D_2 U_2 D_3 U_3 \ldots D_r U_r) \\ & + \mathrm{edit}_1(D_{r+1} U_{r+1} \ldots D_{k-1} U_{k-1} D_k U_k'').\end{aligned}$$

We observe that $\mathrm{edit}_1(D_1', U_k')$ is edit_1 distance between two sequences and we don't do recursion on it. We do recursion on the remaining two parts.

Our recursive algorithm proceeds now as follows. For $t = 1, \ldots, k-1$, we define i_t such that the last symbol of U_t is S_{i_t}. We set $l := \max_t h(i_t)$. By Fact 20, it suffices to consider all parentheses S_i in D_1 such that $|h(i) - l| \leq 10d$. Similarly, it suffices to consider all parentheses S_j in U_k such that $|h(j) - l| \leq 10d$. We conclude that we have $\leq 20d + 1$ choices for i, $\leq 20d + 1$ choices for j and $\leq d$ choices for r. Therefore, if we compute recursively

$$\begin{aligned}M_1 := \min_{\substack{D_1 = D_1' D_1'' \\ U_k = U_k'' U_k' \\ r \in \{1, \ldots, k-1\}}} \ & \mathrm{edit}_1(D_1', U_k') \\ & + \mathrm{edit}_1(D_1'' U_1 D_2 U_2 \ldots D_r U_r) \\ & + \mathrm{edit}_1(D_{r+1} U_{r+1} \ldots D_{k-1} U_{k-1} D_k U_k''),\end{aligned} \tag{3}$$

we make at most $2 \cdot (20d+1) \cdot (20d+1) \cdot d \leq O(d^3)$ different recursive calls (one for each triple i, j, r).

Case 3 We consider this case if one of the following two things happen.

- D_1 or U_k is empty.
- There is an optimal alignment where there are no symbols matched between D_1 and U_k (one from D_1 and one from U_k).

The analysis is similar to Case 2 but simpler. Fix any optimal alignment. We have similar lemma as in Case 2.

Lemma 24. *There exists $r \in \{1, \ldots, k-1\}$ with the following property. There is no symbol in any of the sequences $D_1, D_2, D_3, D_4, \ldots, D_r$ that is matched to a symbol from any of the sequences*

$$U_{r+1}, \ldots, U_{k-3}, U_{k-2}, U_{k-1}, U_k.$$

Proof. The proof is analogous to the proof of Lemma 23. □

Using Lemma 24, we can split the problem into two independent parts for some r. We have that

$$\begin{aligned}\mathrm{edit}_1(S) = \;& \mathrm{edit}_1(D_1U_1D_2U_2 \ldots D_rU_r) \\ &+ \mathrm{edit}_1(D_{r+1}U_{r+1} \ldots D_{k-1}U_{k-1}D_kU_k).\end{aligned}$$

We recurse on the two subproblems. There are at most d choices for r. Therefore, if we compute recursively

$$\begin{aligned}M_2 := \min_r \;& \mathrm{edit}_1(D_1U_1D_2U_2 \ldots D_rU_r) \\ &+ \mathrm{edit}_1(D_{r+1}U_{r+1} \ldots D_{k-1}U_{k-1}D_kU_k),\end{aligned}$$

where we make at most $2d$ recursive calls.

We also notice that, similarly as in Case 2, by Fact 20, if the height difference between the first symbol of S and the last symbol of U_r is $\geq 10d$, we don't need to consider subcall $\mathrm{edit}_1(D_1U_1D_2U_2 \ldots D_rU_r)$. Also, if the height difference between the last symbol of S and the first symbol of D_{r+1} is $\geq 10d$, we don't need to consider subcall $\mathrm{edit}_1(D_{r+1}U_{r+1} \ldots D_{k-1}U_{k-1}D_kU_k)$. We will use this property below.

This finishes the description of the algorithm and Cases 1, 2 and 3.

The correctness of the algorithm follows from the discussions in the description of Cases 1, 2 and 3. We see that the runtime of the algorithm is exponential because we make many recursive calls. However, we can easily fix this by noticing that the number of *different* recursive calls is small. We see that for every subproblem, sequence S starts or ends with a symbol that is in a peak of the original sequence. There are at most $d+1$ peaks in the original problem. For every peak, there are two possibilities: a subproblem can start or end at this peak.

The subproblem ends at the peak The subproblem can start in any of at most d sequences $D_1, D_2, \ldots$ of the original problem. In any of D_i, the subproblem can start with any of at most $20d+1$ symbols.

The subproblem starts at the peak The subproblem can end in any of at most d sequences $U_k, D_{k-1}, \ldots$ of the original problem. In any of U_i, the subproblem can start with any of at most $20d+1$ symbols.

Over all, we see that there are at most $(d+1) \cdot 2 \cdot d \cdot (20d+1) \leq O(d^3)$ subproblems of the original problem to solve. Whenever we solve a subproblem, we store the answer to it so that, when the same subproblem is queried later, we don't need to recompute it. Every subproblem requires

$O(d^3)$ subcalls (see the description of Cases 2 and 3). For a subcall we might need to compute $\text{edit}_1(D_1, U_1)$ (which takes $O(n^2)$ time), if we happen to be in Case 1 or we might possibly need to compute $\text{edit}_1(D'_1, U'_k)$ (which takes $O(n^2)$ time), if we happen to be in Case 2. We get that the final running time is $O(d^3) \cdot O(d^3) \cdot O(n^2) \leq O(n^2 d^6)$ as required.

3.2.2 Second improvement

In the next theorem we improve the runtime of the algorithm described in Theorem 22.

Theorem 25. *Let S be a sequence of parentheses that satisfy Property 19. Assume that* $\text{edit}_1(S) \leq d$. *Then we can compute* $\text{edit}_1(S)$ *in time* $O(d^3(n^2 + d^3))$.

The algorithm described in Theorem 22 considers three cases. If a subproblem falls in Case 1, we can solve it in time $O(n^2)$. Since there are at most $O(d^3)$ subproblems, the total runtime corresponding to Case 1 is upper bounded by $O(d^3 n^2) \leq O(d^3(n^2 + d^3))$. If a subproblem falls in Case 3, then it requires time $O(d)$ (choosing r). Thus, the total runtime corresponding to Case 3 can be upper bounded by $O(d^4) \leq O(d^3(n^2 + d^3))$.

Now we modify the algorithm corresponding to Case 2. Consider a subproblem, where we are given sequence $S = D_1 U_1 \dots D_k U_k$. We compute dynamic programming table for problem $\text{edit}_1(D_1, U_k)$, which takes time $O(n^2)$. This allows us to answer queries $\text{edit}_1(D'_1, U'_k)$ in time $O(1)$ (by single look-up in the dynamic programming table). Since there are at most $O(d^3)$ subproblems, the total time corresponding to Case 2 is $O(d^3)(O(n^2) + O(d^3))$.

We see that in all three Cases 1,2 and 3, the total runtime is upper bounded by $O(d^3(n^2 + d^3))$ as required.

Notice that the runtime bottleneck are cases 1 and 2. In the final improvement below we use the result of Landau et al. [LMS98] to preprocess the input sequence and improve the corresponding runtimes.

3.2.3 Final improvement

In the next theorem we improve the runtime of the algorithm described in Theorem 25.

Theorem 26. *Let S be a sequence of parentheses. Assume that* $\text{edit}_1(S) \leq d$. *Then we can compute* $\text{edit}_1(S)$ *in time* $O(n + d^6)$.

As per discussion above, we can assume that S satisfies Property 19.
We further improve the algorithm presented in Theorem 22.
As was discussed in the proof of Theorem 25, we have:

- The total runtime corresponding to Case 1 is upper bounded by $O(d^3 n^2)$, which we improve to $O(n + d^5) \leq O(n + d^6)$.
- The total runtime corresponding to Case 2 is upper bounded by $O(d^3)(O(n^2) + O(d^3))$, which we will improve to $O(n + d^6)$.
- The total runtime corresponding to Case 3 is upper bounded by $O(d^4) \leq O(n + d^6)$.

Improvement of the total runtime for Case 1 In Case 1 we have to compute $\text{edit}_1(D_1, U_1)$, where D_1 and U_1 are substrings of the original sequence S. Notice that D_1 and U_1 do not necessarily come from the first valley of the original sequence S since the algorithm is recursive. We preprocess S according to Theorem 14. We build the data structure for substring D_1 and U_1 in time $O(d^2)$. From

Theorem 14, we can now answer query $\text{edit}_1(D_1, U_k)$ in $O(\log d)$ time. We had to preprocess the original sequence S but this needs to be done only once. Since we make $O(d^3)$ queries (each taking time $O(d^2) + O(\log d)$), the total runtime bound for Case 1 is

$$O(n) + O(d^3) \cdot \big(O(d^2) + O(\log d)\big) \leq O(n + d^5)$$

as required.

Improvement of the total runtime for Case 2 In Case 2 we have to compute $\text{edit}_1(D_1', U_k')$ many times (see (3)). D_1' is prefix of D_1 and U_k' is suffix of U_k. D_1 and U_k are substrings of the original sequence S. Notice that D_1 and U_k do not necessarily come from the first and the last valley of the original sequence S respectively since the algorithm is recursive. It is not sufficient to preprocess sequences D_1 and U_k in linear time to be able to answer the corresponding queries in $O(1)$. The reason is that we would have to do such preprocessing $d^{O(1)}$ times and the final runtime would be $n \cdot d^{O(1)}$ instead of $O(n + d^{O(1)})$. To avoid this, we preprocess S according to Theorem 14. This allows us to build the data structure for substring D_1 and U_k in time $O(d^2)$ for arbitrary two D_1 and U_k (in the recursive calls). From Theorem 14, we can now answer every query $\text{edit}_1(D_1', U_k')$ in $O(\log d)$ time. We had to preprocess the original sequence S but this needs to be done only once. For every one of $O(d^3)$ subproblems:

- Build a data structure in time $O(d^2)$ for D_1 and U_k.
- Answer at most $(20d + 1)^2$ queries of the form $\text{edit}_1(D_1', U_k')$, each in time $O(\log d)$.
- Explore $O(d^3)$ subcalls.

Thus, the total running time is bounded by

$$O(n) + O(d^3)(O(d^2) + (20d + 1)^2 \cdot O(\log d) + O(d^3))$$
$$= O(n + d^6)$$

as required.

4 Algorithm for distance with substitutions

In this section, we present an algorithm for computing edit distance to a well-balanced sequence of parentheses when both insertions and substitutions are allowed. This section partially mimics the previous one, but also adds another layer of complexity. For instance, when substitutions are allowed, two unrelated opening parentheses, say, "`(`" and "`{`," can be matched after "`{`" is transformed into "`)`." This operation costs one unit, which is better than just deleting both the parentheses, which comes at the higher cost of two. Unfortunately, this shows that now there is no more clear partition into two classes of symbols, opening and closing, with opening always being aligned with closing. Since closing parentheses can be transformed into opening parentheses and vice verse, the number of subproblems (and their handling) grows non-trivially.

4.1 Fast distance computation for a simple case

We start by modifying the subroutine designed in Section 3.1 to the case of substitutions allowed.

Definition 27. *Let X be a sequence of opening parentheses and Y be a sequence of closing parentheses. By* $\text{edit}_2(X, Y)$ *we denote the quantity* $\text{edit}_2(XY)$.

Definition 28. *Let A and B be two sequences of symbols. We define* $\mathrm{edit}_2'(A,B)$ *as the minimum number of operations necessary to transform A and B in to the same sequence, where the allowed operations are*

- *delete a symbol from A or B,*
- *substitute a symbol in A or B with another symbol,*
- *delete* two consecutive *symbols from A or B.*

The next fact is easy to verify by case analysis, which we omit. We just note that the last operation in the definition of edit_2' stems from the following property. For edit_2 with substitutions, if we have two consecutive opening parentheses, we can modify the second (make it closing) to match the first one. Namely, consider the sequences $A =$"((" and $B = \epsilon$ (B is an empty sequence). Then $\mathrm{edit}_2(A,B) = 1$ since we replace the second symbol "(" in A by ")" and we get $AB =$"()" which is a balanced sequence.

Fact 29. $\mathrm{edit}_2(X,Y) = \mathrm{edit}_2'(U(X), \mathrm{rev}(U(Y)))$.

Therefore, to compute $\mathrm{edit}_2(X,Y)$, it suffices to compute $\mathrm{edit}_2'(U(X), \mathrm{rev}(U(Y)))$. We now show that, given two sequences A and B, $\mathrm{edit}_2'(A,B)$ can be computed in $O(n+d^2)$ time, where d upper bounds $\mathrm{edit}_2'(A,B)$ and n upper bounds $|A|$ and $|B|$. To achieve this goal, we modify the algorithm of [LMS98]. [LMS98] gives an algorithm with these guarantess except for a slightly different distance edit_2'. They give an algorithm where deletions of two consecutive symbols are not allowed. (That is, they give algorithm for the *standard* definition of edit distance.) We now describe the changes to the algorithm from [LMS98] that we make.

We show the following variant of Lemma 2.3 from [LMS98].

Lemma 30. *Let A and B be two sequences of symbols. Let a and b be symbols. Then*

$$\mathrm{edit}_2'(Aa, Bb) - \mathrm{edit}_2'(A,B) \in \{0,1\}.$$

Let x be a symbol. Then

$$\mathrm{edit}_2'(Ax, Bx) = \mathrm{edit}_2'(A,B).$$

Proof. We show that $\mathrm{edit}_2'(Ax,Bx) = \mathrm{edit}_2'(A,B)$. Clearly, $\mathrm{edit}_2'(Ax,Bx) \le \mathrm{edit}_2'(A,B)$ since we can match symbols x (which contributes 0 to the cost) and we are left with $\mathrm{edit}_2'(A,B)$. Now we prove that $\mathrm{edit}_2'(Ax,Bx) \ge \mathrm{edit}_2'(A,B)$. If symbols x at the end of Ax and Bx get matched, we get equality. Suppose that a symbol x at the end of Ax or Bx gets deleted. W.l.o.g., symbol x at the end of Ax gets deleted. This costs 1. Now we have

$$\mathrm{edit}_2'(Ax,Bx) = 1 + \mathrm{edit}_2'(A,Bx) \ge \mathrm{edit}_2'(A,B),$$

where in the last inequality we used $\mathrm{edit}_2'(A,Bx) \ge \mathrm{edit}_2'(A,B) - 1$ which is easy to verify. It remains to consider the last case when the last symbol from A gets deleted together with x for unit cost (if the same happens to sequence Bx, the proof is analogous). We write $A = A'y$. We have to show that $\mathrm{edit}_2'(Ax,Bx) = \mathrm{edit}_2'(A'yx,Bx) = 1 + \mathrm{edit}_2'(A',Bx)$ is at least $\mathrm{edit}_2'(A,B)$. Consider the last symbol x in Bx. If it gets deleted, we have

$$\begin{aligned} 1 + \mathrm{edit}_2'(A',Bx) &= 2 + \mathrm{edit}_2'(A',B) \\ &\ge 1 + \mathrm{edit}_2'(A'y,B) = 1 + \mathrm{edit}_2'(A,B), \end{aligned}$$

which suffices for our goal. Suppose that the last symbol x from Bx gets deleted togeher with the last symbol of B. Write $B = B'z$. Then

$$\begin{aligned} 1 + \text{edit}'_2(A', Bx) &= 2 + \text{edit}'_2(A', B') \\ \geq & 1 + \text{edit}'_2(A'y, B'z) = 1 + \text{edit}'_2(A, B), \end{aligned}$$

which is sufficient for our goal. Consider the last operation that can happen to the last symbol x from Bx. It gets aligned to a symbol in A'. Suppose that this x gets to symbol w in A'. Every symbol to the right of w in Ax must be deleted. Let's write $A = A''wA'''$. We have

$$\begin{aligned} \text{edit}'_2(Ax, Bx) &= \text{edit}'_2(A'', B) + \mathit{delete}(A'''x) \\ = & \text{edit}'_2(A'', B) + \mathit{delete}(wA''') \geq \text{edit}'_2(A, B), \end{aligned}$$

which is what we need.

We show that $\text{edit}'_2(Aa, Bb) - \text{edit}'_2(A, B) \in \{0, 1\}$. Clearly, $\text{edit}'_2(Aa, Bb) \leq \text{edit}'_2(A, B) + 1$ since we can substitute a with b. We need to show that $\text{edit}'_2(Aa, Bb) \geq \text{edit}'_2(A, B)$. In the proof that $\text{edit}'_2(Ax, Bx) \geq \text{edit}'_2(A, B)$, we never used the fact that Ax and Bx ends with the same symbol. Therefore, we also have $\text{edit}'_2(Aa, Bb) \geq \text{edit}'_2(A, B)$. □

We replace Lemma 2.8 from [LMS98] with the following lemma.

Lemma 31. *For all $h > 0$,*

$$L^h(d) = \mathit{Slide}_d \left(\max \left\{ \begin{array}{ll} L^{h-1}(d+2)+1 & \text{if } d < h-2 \\ L^{h-1}(d+1)+1 & \text{if } d < h-1 \\ L^{h-1}(d)+1 & \text{if } -h < d < h \\ L^{h-1}(d-1)+1 & \text{if } d > -h+1 \\ L^{h-1}(d-2)+1 & \text{if } d > -h+2 \end{array} \right\} \right).$$

The rest of the algorithm from [LMS98] remains unchanged. We get an algorithm with the following guarantees:

Theorem 32. *We are given two sequences A and B of length at most n. Let d be such that $\text{edit}'_2(A, B) \leq d$. Then we can output $\text{edit}'_2(A, B)$ in $O(n + d^2)$ time.*

We will need the following further consequence of the algorithm from [LMS98].

Theorem 33. *We are given two sequences A and B of length at most n. We can preprocess A and B in $O(n)$ time so that the following holds. Let A' and B' be two substring of A and B, respectively. Let d be such that $\text{edit}'_2(A', B') \leq d$. Then we can in $O(n + d^2)$ time output $\text{edit}'_2(A', B')$.*

Proof. See [LMS98] for the proof. The algorithm in Theorem 32 *preprocesses* the sequences A and B in such a way that we can answer queries $\text{edit}'_2(A', B')$ in time $O(d^2)$.

The only difference between Theorem 32 and this theorem is that we don't know substrings A' and B' of A and B, respectively. However, as in Theorem 12, we only need to be able to answer in constant time queries about longest identical substrings. A data structure answering them can be precomputed in linear time, independently of what A' and B'. □

Now we can get the following theorem that we will use later.

Theorem 34. *We are given sequence S of parentheses of length n. We can preprocess S in time $O(n)$ time so that the following holds. Given any two substrings S_1 and S_2 of S such that* $\mathrm{edit}_2(S_1, S_2) \leq d$ *for some d, we can output* $\mathrm{edit}_2(S_1, S_2)$ *in time $O(d^2)$.*

Proof. We set $A = B = U(S) \circ \mathrm{rev}(U(S))$ and preprocess A and B as in Theorem 33. Now $U(S_1)$ is substring of $U(S)$ and $\mathrm{rev}(U(S_2))$ is substring of $\mathrm{rev}(U(S))$. Since, by Fact 29,

$$\mathrm{edit}_2(S_1, S_2) = \mathrm{edit}'_2(U(S_1), \mathrm{rev}(U(S_2))),$$

we get the required algorithm. □

4.2 Algorithm

Let S be a set of parentheses of length n. Define $A_{i,j} := \mathrm{edit}_2(S_i S_{i+1} \dots S_{j-1} S_j)$. We want to compute $\mathrm{edit}_2(S) = A_{1,|S|}$. Consider the following dynamic programming solution for this problem:

$$A_{i,j} = \min \left\{ \begin{array}{c} A_{i+1,j-1} \text{ if } S_i \text{ and } S_j \text{ aligns,} \\ \min\limits_{r=i,\dots,j-1} A_{i,r} + A_{r+1,j} \end{array} \right\} \tag{4}$$

with base cases $A_{i,i} = 1$ and $A_{i+1,i} = 0$ for all i.

The runtime of this algorithm is $O(n^3)$ and our goal is to obtain an algorithm that runs in time $O(n + d^{16})$, where d upper bounds $\mathrm{edit}_2(S)$. The runtime of the $O(n^3)$ time algorithm comes from the fact that we are examining $O(n^3)$ triples (i, j, r). Our idea is to improve this algorithm by examining only $O(n + d^{16})$ triples. The rest of this section is devoted to implementing this idea with the help of Theorem 34.

Recall Definitions 15, 16, and 17 as well as the decomposition (2) from Section 3. The following claim is an analogue of Claim 21 with a very similar proof.

Claim 35. $k \leq 2d$.

We need the following fact, which is a slight variation of Fact 20.

Fact 36. *Consider subsequence $S_i S_{i+1} \dots S_j$ of S for some integers $j \geq i$. If $|h(j) - h(i)| > 2d$, then*

$$\mathrm{edit}_2(S_i S_{i+1} \dots S_j) > d.$$

The fact follows from the observations that a deletion or insertion change $|h(j) - h(i)|$ by at most 1 and that a substitution change $|h(j) - h(i)|$ by at most 2.

We also need the following definition.

Definition 37. *We call the last symbol of D_1 (if D_1 is nonempty) and the first symbol of U_1 the 1-th base of S. For $i = 2, \dots, k-1$, we call the last symbol of D_i and the first symbol of U_i the i-th base of S. We call the last symbol of D_k and the first symbol of U_k (if U_k is nonempty) the k-th base of S.*

We define set H:

$$\begin{aligned} H := &\{h(i) \mid S_i \text{ belongs to the } j\text{-th peak} \\ &\qquad \text{for some } j = 0, \dots, k\} \\ &\cup \{h(i) \mid S_i \text{ belongs to the } j\text{-th base} \\ &\qquad \text{for some } j = 1, \dots, k\}. \end{aligned} \tag{5}$$

We define set L:

$$L := \{[v - 100d, v + 100d] \mid v \in H\}. \tag{6}$$

We can think of set L as the set of heights where alignments between parentheses between more than two slopes can happen. We will see that for sequences of symbols S_i with $h(i) \notin L$, we can deal fairly easy - by using Theorem 34.

As long as there exist $[a,b],[c,d] \in L$ with $[a,b] \neq [c,d]$ and $[a,b] \cap [c,d] \neq \emptyset$, we update

$$L \leftarrow (L \setminus \{[a,b],[c,d]\}) \cup \{[a,b] \cup [c,d]\}.$$

That is, we merge any two intersecting intervals into one. Let $L = \{[a_1,b_1],\ldots,[a_l,b_l]\}$ be the resulting set L for $a_1 < b_1 < a_2 < b_2 < \ldots < a_l < b_l$. We call $[a_t,b_t] \in L$ the t-th *layer* of S. Since the total number of peaks and bases is upper bounded by $2d+1$, the number l of layers is upper bounded by $l \leq 2d+1$. Also, we have

$$|[a_t,b_t]| = b_t - a_t + 1 \leq (200d+1) \cdot l + 1 \leq O(d^2).$$

That is, the length of every layer is upper bounded by $O(d^2)$.

We define the following set of pairs of indices:

$$E := \{(i,j) \mid \exists t: \ h(i), h(j) \in [a_t,b_t]\}.$$

Since the number of valleys is $k \leq 2d$ (by Claim 35), we get

$$|E| \leq \left(\sum_{t=1}^{l} (k \cdot |[a_t,b_t]|)\right)^2 \leq O(d^8).$$

Now let's reconsider Algorithm (4). The idea is to preprocess S in $O(n)$ time so that we can consider only triples (i,j,r) that satisfy $(i,j),(i,r),(j,r) \in E$. There are at most $O(|E|^2)$ such triples. Additionally, per every tuple (i,j), we will have to spend time $O(d^4)$ (see the algorithm below). There are at most $O(|E|)$ such tuples. The total runtime, including the preprocessing is

$$O(n) + O(|E|^2) + O(|E|) \cdot O(d^4) \leq O(n + d^{16}),$$

which is what we wanted.

To implement the new algorithm, we need the following definitions.

Definition 38. *Let $[a_t,b_t] \in L$ be the t-th layer. We call $[a_t, a_t+10d]$ the* bottom *of the t-th layer. We call $[b_t-10d, b_t]$ the* top *of the t-th layer.*

Definition 39. *We call i and j to be* top neighbors *(*bottom neighbors*, respectively) in layer t if the following conditions hold:*

- *$i < j$.*
- *$h(i)$ and $h(j)$ are both in the top (bottom, respectively) of the t-th layer.*
- *S_i is a symbol in D_r for some r and S_j is a symbol in $U_{r'}$ for some r'. Consider smallest $j' > i$ such that $h(j')$ belongs to the top (bottom, resp.) of t-th layer and $S_{j'}$ belongs to $U_{r''}$ for some r''. Then we have property that $r' = r''$.*

Now we can describe our recursive algorithm that is based on Algorithm (4) and that runs in time $O(n+d^{16})$.

Before running the recursive algorithm, we first preprocess S in $O(n)$ time according to Theorem 34. Then we run the recursive algorithm:

1. We are given $(i, j) \in E$ with $i < j$. If we already computed $A_{i,j}$ before, we return the previously computed answer.

2. If i and j are *not* bottom neighbors in any layer t, then:

   (a) For $r = i, \ldots, j-1$, define M_r as follows. If $(i, r) \notin E$ or $(r+1, j) \notin E$, set $M_r = +\infty$. Otherwise, recursively compute $A_{i,r}$ and $A_{r+1,j}$ and set $M_r = A_{i,r} + A_{r+1,j}$.

   (b) Set $M := \min_{r=i,\ldots,j-1} M_r$.

   (c) If parentheses S_i and S_j align, recursively compute $A_{i+1,j-1}$ and set $M' := A_{i+1,j-1}$. Otherwise, set $M' := +\infty$.

   (d) return $\min(M, M')$.

3. Let layer t be such that i and j are bottom neighbors in this layer. By the definition of L, there must be i' and j' that satisfy:

   (a) i' and j' are top neighbors in layer $t-1$.

   (b) $i < i' < j' < j$.

   There are at most $(10d+1)^2 \leq O(d^2)$ such tuples (i', j') by Definition 39. Because of the preprocessing and Theorem 34, we can compute

   $$\mathrm{edit}_2(S_i \ldots S_{i'-1}, S_{j'+1} \ldots S_j)$$

   in time $O(d^2)$.

   For every i', j', we recursively compute $A_{i',j'}$ and set

   $$M_{i',j'} := \mathrm{edit}_2(S_i \ldots S_{i'-1}, S_{j'+1} \ldots S_j) + A_{i',j'}.$$

4. Return $\min_{i',j'} M_{i',j'}$.

Output $\mathrm{edit}_2(S)$ is equal to $A_{1,|S|}$. $(1, |S|) \in E$ holds by the definition of E and L and the fact that $h(1) - h(|S|) \leq 2d$ (remember assumption that $\mathrm{edit}_2(S) \leq d$ and see Claim 35).

The runtime analysis Preprocessing is done in time $O(n)$. The total time corresponding to Step 2 is $O(|E|^2)$. This holds due to at most $|E|$ choices for tuple (i, j) and, for every choice of tuple (i, j), there are at most $|E|$ choices for r. The total runtime corresponding to Step 3 is $O(|E|d^4)$. This is so because there are $O(|E|)$ choices for tuple (i, j). And for every choice of tuple we have at most $O(d^2)$ choices of tuple (i', j'). And for every choice of i, j, i', j', we must do computational task taking $O(d^2)$ time (computing an instance of edit_2 problem). Thus, the total runtime is:

$$O(n) + O(|E|^2) + O(|E|) \cdot O(d^2) \cdot O(d^2) \leq O(n + d^{16})$$

as required.

The correctness of the algorithm We prove correctness inductively. We show that we compute $A_{i,j}$ correctly, if the recursive calls that we make, all return correct answers. We consider two cases.

***i* and *j* are such that we do Step 2** i and j are not bottom neighbors in any layer t. By definition of set E, Definition 38 and Fact 36, we have that it is sufficient to consider all $r = i, \ldots, j-1$ for which $(i, r), (r+1, j) \in E$. Correctness for this step follows from the correctness of Algorithm 4.

i and j are such that we do Step 3 i and j are bottom neighbors in layer t. Consider an optimal alignment for $S_i \dots S_j$ achieving cost $\mathrm{edit}_2(S_i \dots S_j)$. There must be i' and j' with $i < i' < j' < j$ such that

$$\begin{aligned}\mathrm{edit}_2(S_i \dots S_j) = \ & \mathrm{edit}_2(S_i \dots S_{i'-1}, S_{j'+1} \dots S_j) \\ & + \mathrm{edit}_2(S_{i'} \dots S_{j'}).\end{aligned}$$

This is so because $h(i)$ and $h(j)$ belong to bottom of layer t. We can choose i' and j' so that i' and j' are top neighbors in layer $t-1$. This follows from Definitions 38 and 39.

This finishes the correctness proof.

We get the following theorem.

Theorem 40. *Let S be a sequence of parentheses. In $O(n + d^{16})$ time we can compute* $\mathrm{edit}_2(S)$.

5 Acknowledgments

We thank Piotr Sankowski for early discussions. We thank Piotr Indyk for many helpful comments on the early version of the write-up and suggesting the title. We also thank anonymous reviewers for many helpful comments on how to improve the presentation. Arturs Backurs was supported by NSF and the Simons Foundation.

References

[ABVW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. *FOCS*, 2015.

[AP72] Alfred V Aho and Thomas G Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972.

[BI15] 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 2015, Portland, OR, USA, June 14-17, 2015*, pages 51–58, 2015.

[Gal14] François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303, 2014.

[GM97] Zvi Galil and Oded Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134(2):103 – 139, 1997.

[htm] WHATWG's HTML Living Standard. `https://html.spec.whatwg.org/`. Accessed: 2015-12-03.

[KSSY13] Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. *Proceedings of the VLDB Endowment*, 6(9):601–612, 2013.

[LMS98] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, April 1998.

[Mye95] Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995.

[Sah14] Barna Saha. The Dyck language edit distance problem in near-linear time. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 611–620. IEEE, 2014.

[Sah15] Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 118–135. IEEE, 2015.

[SZ99] Avi Shoshan and Uri Zwick. All pairs shortest paths in undirected graphs with integer weights. In *In IEEE Symposium on Foundations of Computer Science*, pages 605–614, 1999.

[Val75] Leslie G Valiant. General context-free recognition in less than cubic time. *Journal of computer and system sciences*, 10(2):308–315, 1975.

[VGF14] Balaji Venkatachalam, Dan Gusfield, and Yelena Frid. Faster algorithms for rna-folding using the four-russians method. *Algorithms for Molecular Biology*, 9(1):1, 2014.

[Wil12] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th symposium on Theory of Computing*, pages 887–898. ACM, 2012.

[Wil14] Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 664–673. ACM, 2014.