The Dyck Language Edit Distance Problem in Near-linear Time

Barna Saha
Dept. of Computer Science
University of Massachusetts, Amherst
Amherst, USA
Email: barna@cs.umass.edu.

Abstract

Given a string σ over alphabet Σ and a grammar G defined over the same alphabet, how many minimum number of repairs (insertions, deletions and substitutions) are required to map σ into a valid member of G? The seminal work of Aho and Peterson in 1972 initiated the study of this *language edit distance problem* providing a dynamic programming algorithm for context free languages that runs in $O(|G|^2n^3)$ time, where n is the string length and $|G|$ is the grammar size. While later improvements reduced the running time to $O(|G|n^3)$, the cubic running time on the input length held a major bottleneck for applying these algorithms to their multitude of applications.

In this paper, we study the language edit distance problem for a fundamental context free language, $\text{DYCK}(s)$ representing the language of well-balanced parentheses of s different types, that has been pivotal in the development of formal language theory. We provide the very first *near-linear time* algorithm to tightly approximate the $\text{DYCK}(s)$ language edit distance problem for any arbitrary s. $\text{DYCK}(s)$ language edit distance *significantly generalizes* the well-studied *string edit distance problem*, and appears in most applications of language edit distance ranging from data quality in databases, generating automated error-correcting parsers in compiler optimization to structure prediction problems in biological sequences. Its nondeterministic counterpart is known as the hardest context free language.

Our main result is an algorithm for edit distance computation to $\text{DYCK}(s)$ for any positive integer s that runs in $O(n^{1+\epsilon}polylog(n))$ time and achieves an approximation factor of $O(\frac{1}{\epsilon}\beta(n)\log|OPT|)$, for any $\epsilon > 0$. Here OPT is the optimal edit distance to $\text{DYCK}(s)$ and $\beta(n)$ is the best approximation factor known for the simpler problem of string edit distance running in analogous time. If we allow $O(n^{1+\epsilon} + |OPT|^2n^{\epsilon})$ time, then the approximation factor can be reduced to $O(\frac{1}{\epsilon}\log|OPT|)$. Since the best known near-linear time algorithm for the string edit distance problem has $\beta(n) = polylog(n)$, under near-linear time computation model both $\text{DYCK}(s)$ language and string edit distance problems have $polylog(n)$ approximation factors. This comes as a surprise since the former is a significant generalization of the latter and their exact computations via dynamic programming show a stark difference in time complexity.

Rather less surprisingly, we show that the framework for efficiently approximating edit distance to $\text{DYCK}(s)$ can be utilized for many other languages. We illustrate this by considering various memory checking languages (studied extensively under distributed verification) such as STACK, QUEUE, PQ and DEQUE which comprise of valid transcripts of stacks, queues, priority queues and double-ended queues respectively. Therefore, any language that can be recognized by these data structures, can also be repaired efficiently by our algorithm.

1 Introduction

Given a string σ over alphabet Σ and a grammar G defined over the same alphabet, how many minimum number of repairs (insertions, deletions and substitutions) are required to map σ into a valid member of G? In this work, we consider such *language edit distance problem* with respect to $\text{DYCK}(s)$, where $|\Sigma|= 2s$.

$\textsc{Dyck}(s)$ is a fundamental context free grammar representing the language of well-balanced parentheses of s different types, and $\textsc{Dyck}$ language edit distance is a *significant generalization* of the *string edit distance problem* which has been studied extensively in theoretical computer science and beyond.

Dyck language appears in many contexts. These languages often describe a property that should be held by commands in most commonly used programming languages, as well as various subsets of commands/symbols used in Latex. Variety of semi-structured data from XML documents to JSON data interchange files to annotated linguistic corpora contain open and close tags that must be properly nested. They are frequently massive in size and exhibit complex structures with arbitrary levels of nesting tags (an XML document often encodes an entire database). For example, dblp.xml has current size of 1.2 GB, is growing rapidly, with 2.3 million articles that results in a string of parentheses of length more than 23 million till date. In addition, Dyck language plays an important role in DNA evolutionary languages and RNA structure modeling where the base nucleotide pairs in DNA/RNA sequences need to match up in a well-formed way. Deviations from this well-formed matching reveal interesting properties of the underlying biological sequences [17, 32]. Dyck language has been pivotal in the development of the theory of context-free languages (CFL). As stated by the Chomsky-Schotzenberger Theorem, every CFL can be mapped to a restricted subset of $\textsc{Dyck}(s)$ [10]. A comprehensive description of context free languages and Dyke languages can be found in [18, 21].

The study of *language edit distance problem* dates back to early seventies. Such an algorithm for context free grammar (CFG) was first proposed by Aho and Peterson that runs in $O(|G|^2n^3)$ time where $|\sigma|= n$ is the string length and $|G|$ is the size of the grammar [1]. This was later improved by Myers to $O(|G|n^3)$ time [28]. These works were motivated by developing automated parsers for compiler design. For $\textsc{Dyck}(s)$ a dynamic programming algorithm gives a running time of $O(n^3)$ independent of s. Since a well-balanced string may be composed of two well-balanced substrings, the optimum edit distance computation for a substring from index i to j, $1 \le i < j \le n$, requires checking all possible decompositions of the substring at intermediate indices $k = i, i+1, ..., j$. This leads to the cubic dependency in the running time. It is possible to improve this bound slightly to $O(\frac{n^3}{\log n})$ using the Four-Russian technique [35].

Nearly two decades back, [27] reported these algorithms with cubic running time to be prohibitively slow for parser design. With modern data deluge, the issue of scalability has become far more critical. Motivated by a concrete application of repairing semi-structured documents where imbalanced parenthesis nesting is one of the major reported errors (14% of XML errors on the web is due to malformedness [16]) and lack of scalability of cubic time algorithms, the authors in [20] study the problem of approximating edit distance computation to $\textsc{Dyck}(s)$. Given any string σ if $\sigma' = \min_{x \in Dyck(s)} StrEdit(\sigma, x)$, they ask the question whether it is possible to design an algorithm that runs in near-linear time and returns $\sigma'' \in Dyck(s)$ such that $StrEdit(\sigma, \sigma'') \le \alpha StrEdit(\sigma, \sigma')$ for some $\alpha \ge 1$ where $StrEdit$ is the normal string edit distance function and α is the approximation factor. Edit distance computation from a string of parentheses to $\textsc{Dyck}(s)$ is a significant generalization of string edit distance computation [1]. A prototypical dynamic programming for string edit distance computation runs in quadratic time (as opposed to cubic time for $\textsc{Dyck}(s)$ edit distance problem). There is a large body of works on designing scalable algorithms for approximate string edit distance computation [4–7, 30]. Though basic in its appeal, nothing much is known for approximately computing edit distance to $\textsc{Dyck}(s)$.

In [20], the authors proposed fast greedy and branch and bound methods with various pruning strategies to approximately answer the edit distance computation to $\textsc{Dyck}(s)$, and showed its applicability in practice

[1]For string edit distance computation, between string σ_1 and σ_2 over alphabet C, create a new alphabet $T \cup \bar{T}$ by uniquely mapping each character $c \in C$ to a new type of open parenthesis, say t_c, that now belongs to T. Let $\bar{t_c}$ be the matching close parenthesis for t_c and we let $\bar{t_c} \in \bar{T}$. Now create strings σ_1' by replacing each character of σ_1 with its corresponding open parenthesis in T, and create string σ_2' by replacing each character of σ_2 with its corresponding close parenthesis in $\bar{T}$. Obtain σ by appending σ_1' with reverse of σ_2'. It is easy to check the edit distance between σ and $\textsc{Dyck}(s)$ is exactly equal to string edit distance between σ_1 and σ_2.

over rule based heuristics commonly employed by modern web browsers like Internet Explorer, Firefox etc. However, either their algorithms have worst-case approximation ratio as bad as $\Theta(n)$ or have running time exponential in $|OPT|$ (see [20] for worst case examples). It is to be noted that for $\text{DYCK}(1)$, there exists a simple single pass algorithm to compute edit distance: just pair up matching open and close parentheses and report the number of parentheses that could not be matched in this fashion.

In this paper, we study the question of approximating edit distance to $\text{DYCK}(s)$ for any $s \geq 2$ and give the *first* near-linear time algorithm with nontrivial approximation guarantees. Our main result is an algorithm for edit distance computation to $\text{DYCK}(s)$ language for any positive integer s that runs in $O(n^{1+\epsilon}polylog(n))$ time and achieves an approximation factor of $O(\frac{1}{\epsilon}\beta(n)\log|OPT|)$, for any $\epsilon > 0$. Here OPT is the optimal edit distance to $\text{DYCK}(s)$ and $\beta(n)$ is the best approximation factor known for the simpler problem of string edit distance running in analogous time. If we allow $O(n^{1+\epsilon} + |OPT|^2 n^{\epsilon})$ time, then the approximation factor can be reduced to $O(\log|OPT|)$. Since the best known near-linear time algorithm for the string edit distance problem has $\beta(n) = polylog(n)$, under near-linear time computation model both the string edit distance and the $\text{DYCK}(s)$ language edit distance problems have $polylog(n)$ approximation factors. This comes as a surprise since edit distance to $\text{DYCK}(s)$ is a significant generalization of the string edit distance problem and their exact computations via dynamic programming show a marked difference in time complexity.

Any parentheses string σ can be viewed as $Y_1X_1Y_2X_2....Y_zX_z$ for some $z \geq 0$ where Y_is and X_is respectively consist of only consecutive open and close parentheses. The special case of string edit distance problem can be cast as having $z = 1$. The approximation factor of our algorithm is in fact depends only on $\log z$, and not $\log|OPT|$. It is possible to ensure $z \leq OPT$ by a simple preprocessing. Our algorithm is based on a judicious combination of random walks in multiple phases that guide selection of subsequences having a single sequence of open parentheses followed by a single sequence of close parentheses. These subsequences are then repaired each by employing subroutine for STREDIT computation. The general framework of our algorithm and its analysis applies to languages far beyond $\text{DYCK}(s)$. We discuss this connection with respect to the memory checking languages whose study was initiated in the seminal work of Blum, Evans, Gemmell, Kannan and Naor [8] with numerous follow-up works [2, 9, 11, 13, 29]. We consider basic languages such as STACK, QUEUE, PQ, DEQUE etc. They comprise of valid transcripts of stacks, queues, priority queues and double-ended queues respectively. Given a transcript of any such memory-checking language, we consider the problem of finding the minimum number of edits required to make the transcript error-free and show that the algorithm for $\text{DYCK}(s)$ can be adapted to return a valid transcript efficiently with the same approximation bound. Therefore, any language that can be recognized by these data structures, can also be repaired efficiently by our algorithm. We believe our novel multi-phase random walk based technique can be useful for any generic sequence alignment type problems and may lead to a systematic way of speeding up dynamic programming algorithms for this large class of problems.

1.1 Related Work

Early works on edit distance to grammar [1, 28] was motivated by the problem of correcting and recovering from syntax error during context-free parsing and have received significant attention in the realm of compiler optimization [14, 15, 19, 27]. Many of these works focus on designing time-efficient parsers using local recovery [15, 19, 27] rather than global dynamic programming based algorithms [1, 28], but to the best of our knowledge, none of these methods provide approximation guarantee on edit distance in sub-cubic time. Approximating edit distance to $\text{DYCK}(s)$ has recently been studied in [20] for repairing XML documents, and in [35] for RNA folding. But either the proposed algorithms have running time very close to cubic [35], or the proposed subcubic algorithms all have worst case approximation factor $\Theta(n)$ [20].

Recognizing a grammar is a much simpler task than repairing, and DYCK language recognition has attracted considerable attention before. Using a stack, it is straightforward to recognize $\text{DYCK}(s)$ in a single pass-a prototypical example of a stack-based algorithm. When there is a space restriction, Magniez, Mathieu and Nayak [26] considered the streaming complexity of recognizing $\text{DYCK}(s)$ showing an $\Omega(\sqrt{n})$ lower bound and a matching upper bound within a $\log n$ factor. Even with multiple one-directional passes, the lower bound remains at $\Omega(\sqrt{n})$ [9], surprisingly with two passes in opposite directions the space complexity reduces to $O(\log^2 n)$. This exhibits a curious phenomenon of streaming complexity of recognizing $\text{DYCK}(s)$. Krebs et al. extended the work of [26] to consider very restricted cases of errors, where an open (close) parenthesis can only be changed into another open (close) parenthesis [22]. Again with only such edits, computing edit distance in linear time is trivial: whenever an open parenthesis at the stack top cannot be matched with the current close parenthesis, change one of them to match. Allowing arbitrary insertions, deletions and substitutions is what makes the problem significantly harder. In the property testing framework, in the seminal paper Alon, Krivelevich, Newman and Szegedy [3] showed that $\text{DYCK}(1)$ is testable in time independent of n, however $\text{DYCK}(2)$ requires $\Omega(\log n)$ queries. This lower bound was further strengthened to $n^{1/11}$ by Parnas, Ron and Rubinfeld [31] where they also give a testing algorithm using $n^{2/3}/\epsilon^3$ queries. These algorithms can only distinguish between the case of 0 error with ϵn errors, and therefore, are not applicable to the problem of approximating edit-distance to $\text{DYCK}(s)$.

Edit distance to $\text{DYCK}(s)$ is a significant generalization of string edit distance problem. String edit distance enjoys the special property that if the optimal edit distance is d then a symbol at the ith position in one string must be matched with a symbol at a position between $(i-d)$ to $(i+d)$ in the other string, if d is the optimum string edit distance. Using this "local" property, prototypical quadratic dynamic programming algorithm can be improved to run in time $O(dn)$ which was latter improved to $O(n+d^7)$ [33] to $O(n+d^4)$ [12] and then to $O(n+d^2)$ [23]. The later result implies a $\sqrt{n}$-approximation for string edit distance problem. However, all of these crucially use the locality property, which does not hold for parenthesis strings: two parentheses far apart can match as well. Also, it is known that parsing arbitrary CFG is as hard as boolean matrix multiplication [24] and a nondeterministic version of DYCK is the hardest CFG [10]. Therefore, exactly computing edit distance to $\text{DYCK}(s)$ in time much less than subcubic would be a significant accomplishment. For string edit distance, the current best approximation ratio of $(\log n)^{O(\frac{1}{\epsilon})}$ in $O(n^{1+\epsilon} polylog(n))$ running time for any fixed $\epsilon > 0$ is due to Andoni, Krauthgamer and Onak [4]. This result is preceded by a series of works which improved the approximation ratio from $\sqrt{n}$ [23] to $n^{\frac{3}{7}}$ [6], then to $n^{\frac{1}{3}+o(1)}$ [7], all of which run in linear time and finally to $2^{\sqrt{\log n \log\log n}}$ that run in time $n2^{\sqrt{\log n \log\log n}}$ [5].

1.2 Techniques & RoadMap

Definition 1. *The* congruent *of a parenthesis x is defined as its symmetric opposite parenthesis, denoted $\bar{x}$. The congruent of a set of parentheses X, denoted $\bar{X}$, is defined as $\{\bar{x} \mid x \in X\}$.*

We use T to denote the set of *open parentheses* and $\bar{T}$ to denote the set of *close parentheses*. (Each $x \in T$ has exactly one congruent $\bar{x} \in \bar{T}$ that it matches to and vice versa.) The alphabet $\Sigma = T \cup \bar{T}$.

Definition 2. *A string over some parenthesis alphabet $\Sigma = T \cup \bar{T}$ is called* well-balanced *if it obeys the context-free grammar* $\text{DYCK}(s)$*, $s = |T|$ with productions $S \to SS$, $S \to \phi$ (empty string) and $S \to aS\bar{a}$ for all $a \in T$.*

Definition 3. *The* DYCK Language Edit Distance Problem*, given string $\sigma = \sigma_1 ... \sigma_n$ over alphabet $\Sigma = T \cup \bar{T}$, is to find* $\arg\min_{\sigma'} StrEdit(\sigma, \sigma')$ *such that $\sigma' \in \text{DYCK}(s)$, $s = |T|$.*

A Simple Algorithm (Section 2)

We start with a very simple algorithm that acts as a stepping stone for the subsequent refinements. The algorithm is as simple as it gets, and is referred to as **Random-deletion**.

Initialize a stack to empty. Scan the parenthesis string σ left to right. If the current symbol is an open parenthesis, insert it into the stack. If the current symbol is a close parenthesis, check the symbol at top of the stack. If both the symbols can be matched, match them. If the stack is empty, delete the current symbol. Else **delete one of them independently with equal probability** $\frac{1}{2}$. *If the stack is nonempty when the scan has ended, delete all symbols from the stack.*

Let d be the optimum edit distance to $\text{DYCK}(s)$. Each deletion accounts for one edit. We show

Theorem 1. **Random-deletion** *obtains a $4d$-approximation for edit distance computation to* $\text{DYCK}(s)$ *for any $s \geq 2$ in linear time with constant probability.*

The probability can be boosted by running the algorithm $\Theta(\log n)$ times and considering the iteration which results in the minimum number of edits. In the worst case, when $d = \sqrt{n}$, the approximation factor can be $4\sqrt{n}$. This also gives a very simple algorithm for string edit distance problem that achieves a $O(\sqrt{n})$ approximation.

The analysis of even such a simple algorithm is nontrivial and proceeds as follows. First, we allow the optimal algorithm to consider only deletion and allow it to process the string using a stack; this increases the edit distance by a factor of 2 (Lemma 6, Lemma 7). We then define for each comparison where an open and close parenthesis cannot be matched, a corresponding correct and wrong move. If an optimal algorithm also compares exactly the two symbols and decides to delete one, then we can simply define the parenthesis that is deleted by the optimal algorithm as the correct move and the other as a wrong move. However, after the "first" wrong move, the comparisons performed by our procedure vs the optimal algorithm can become very different. Yet, we can label one of the two possible deletions as a correct move in some sense of decreasing distance to an optimal state. Consider a toy example,$cb\bar{a}\bar{b}\bar{c}$. An optimal algorithm deletes $\bar{a}$ to make the string well-formed. The first comparison performed by our algorithm may make a wrong move and delete b instead of $\bar{a}$. At the next comparison between c and $\bar{a}$, it still can recover from this wrong move without paying too much cost in the edit distance if it deletes $\bar{a}$, which is the correct move in this case. We show that if up to time t, the algorithm has taken W_t wrong moves then it is possible to get a well-formed string using a total of $2(d + 2W_t)$ edit operations. These two properties help us to map the process of deletions to a one dimensional random walk which is known as the GAMBLER'S RUIN problem.

In the considered gambler's ruin problem, a gambler enters a casino with $\$d$ (remember d is the optimum edit distance to $\text{DYCK}(s)$) and starts playing a game where he wins with probability $1/2$ and loses with probability $1/2$ independently. The gambler plays the game repeatedly betting \$1 in each round. He leaves if he runs out of money or gets $\$n$. We can show that the number of edit operations performed by our algorithm can not be more than the number of steps taken by the gambler to be ruined. However, on expectation, the gambler takes $O(n)$ steps to be ruined, and this bound is not useful for our purpose. Interestingly, the underlying probability distribution of the number of steps taken by the gambler is heavy-tailed and using that property, one can still show that there is considerable probability ($\sim \frac{1}{5}$) that gambler is ruined in $O(d^2)$ steps. Therefore the total number of edits of our algorithm is bounded by $O(d^2)$ leading to an $O(d)$ approximation.

A Refined Algorithm (Section 3)

We now refine our algorithm as follows. Given string σ, we can delete any prefix of close parentheses, delete any suffix of open parentheses and match well-formed substrings without affecting the optimal solution. After that, σ can be written as $Y_1X_1Y_2X_2....Y_zX_z$ where each Y_i is a sequence of open parentheses, each X_i is a sequence of close parentheses and $z \leq d$. In the optimal solution X_1 is matched with some suffix of Y_1 possibly after doing edits. Let us denote this suffix by Z_1. If we can find the left boundary of Z_1, then we can employ STREDIT to compute string edit distance between Z_1 and X_1 (we need to consider reverse of X_1

and convert each $t \in X_1$ to $\bar{t}$–this is what is meant when we refer STREDIT between a sequence of open and a sequence of close parentheses), as Z_1X_1 consists of only a single sequence of open parentheses followed by a single sequence of close parentheses. If we can identify Z_1 correctly, then in the optimal solution X_2 is matched with a suffix of $Y_1^{res}Y_2$ where $Y_1^{res} = Y_1 \setminus Z_1$. Let us denote it by Z_2. If we can again find the left boundary of Z_2, then we can employ STREDIT between Z_2 and X_2 and so on. The question is *how do we compute these boundaries?*

The key trick is to use *Random-deletion* again (repeat it appropriately $\sim \log n$ times) to find these boundaries approximately (see Algorithm 8.1). We consider the suffix of Y_1 which *Random-deletion* matches against X_1 possibly after multiple deletions (call it Z_1') and use Z_1' to approximate Z_1. We show again using the mapping of *Random-deletion* to random walk, that the error in estimating the left boundary is bounded (Lemma 9, Lemma 17, Lemma 18). Specifically, if $StrEdit(Z_1, X_1) = d_1$, then the error in estimating the boundaries is at most $d_1\sqrt{2\log d_1}$ and $StrEdit(Z_1', X_1) \leq 2d_1\sqrt{2\ln d_1}$. But, the error that we make in estimating Z_1 may propagate and affect the estimation of Z_2. Hence the gap between optimal Z_2 and estimated Z_2' becomes wider. If $StrEdit(Z_2, X_2) = d_2$, then we get $StrEdit(Z_2', X_2) \leq 2(d_1+d_2)\sqrt{2\ln{(d_1+d_2)}}$. Proceeding, in this fashion, we get the following theorem.

Theorem 2. *Algorithm 8.1 obtains an $O(z\beta(n)\sqrt{\ln d})$-approximation factor for edit distance computation from strings to* DYCK(s) *for any $s \geq 2$ in $O(n\log n + \alpha(n))$ time with probability at least $\left(1 - \frac{1}{n} - \frac{1}{d}\right)$, where there exists an algorithm for* STREDIT *running in $\alpha(n)$ time that achieves an approximation factor of $\beta(n)$.*

Further Refinement: Main Algorithm (Section 4)

Is it possible to compute subsequences of σ such that each subsequence contains a single sequence of open and close parentheses in order to apply STREDIT*, yet propagational error can be avoided?* This leads to our main algorithm.

Example. Consider $\sigma = Y_1X_1Y_2X_2Y_3X_3...Y_zX_z$, and let the optimal algorithm matches X_1 with $Z_{1,1}$, matches X_2 with $Z_{1,2}Y_2$, matches X_3 with $Z_{1,3}Y_3$, and so on, where $Y_1 = Z_{1,z}Z_{1,z-1}...Z_{1,2}Z_{1,1}$. In the *refined algorithm*, when *Random-deletion* finishes processing X_1 and tries to estimate the left boundary of $Z_{1,1}$, it might have already deleted some symbols of $Z_{1,2}$. It is possible that it deletes $\Theta(d_1\sqrt{\log d_1})$ symbols from $Z_{1,2}$. Therefore, when computing STREDIT between X_1 and Z_1', the portion of $Z_{1,2}$ in Z_1' may not have any matching symbols and results in an increased STREDIT computation. More severely, the *Random-deletion* process gets affected when processing X_2. *Random-deletion* does not find the already deleted symbols in $Z_{1,2}$ which ought to be matched with some subsequence of X_2. As a result, it may start comparing $Z_{1,3}$ with X_2, and in the process may delete up to $\Theta((d_1+d_2)\sqrt{\log{(d_1+d_2)}})$ symbols from $Z_{1,3}$, and so on. To remedy this, view $X_2 = X_{2,in}X_{2,out}$ where $X_{2,in}$ is the prefix of X_2 that is matched with Y_2 and $X_{2,out}$ is matched with $Z_{1,2}$. Consider pausing the random deletion process when it finishes Y_2 and thus attempt to find $X_{2,in}$. Suppose, *Random-deletion* matches $X_{2,in}'$ with Y_2, then compute $StrEdit(Y_2, X_{2,in}')$. While there could still be a mistake in computing $X_{2,in}'$, the mistake does not affect $Z_{1,3}$. Else if *Random-deletion* process finishes X_2 before finishing Y_2, then of course it affected $Z_{1,3}$. In that case Z_2' is a suffix of Y_2 and we compute $StrEdit(Z_2', X_2)$. Similarly, when processing X_3, we pause whenever X_3 or Y_3 is exhausted and create an instance of STREDIT accordingly. Suppose, for the sake of this example, $Y_2, Y_3, .., Y_z$ are finished before finishing $X_2, X_3, ..., X_z$ respectively and X_1 is finished before Y_1. Then, after creating the instances of STREDITs as described, we are left with a sequence of open parenthesis corresponding to a prefix of Y_1 and a sequence of close parenthesis, which is a combination of suffixes from $X_2, X_3, ..., X_z$. We can compute STREDIT between them. Since most of $Z_{1,z}Z_{1,z-1}...Z_{1,2}$ exists in this remaining prefix of Y_1 and their matching parentheses in the constructed sequence of close parentheses, the computed STREDIT distance remains small.

Let us call each X_iY_i a block. As the example illustrates, we create StrEdit instances corresponding to what *Random-deletion* does *locally* inside each block. After the first phase, from each block we are either left with a sequence of open parentheses (call it a block of type O), or a sequence of close parentheses (call it a block of type C), or the block is empty. This creates new sequences of open and close parentheses by combining all the consecutive O blocks together (remove empty blocks) and similarly combining all consecutive C blocks together (remove empty blocks). We get at most $\lfloor \frac{z}{2} \rfloor$ new blocks in the remaining string after deleting any prefix of close and any suffix of open parentheses. We now repeat the process on this new string. This process can continue at most $\lceil \log z \rceil + 1$ phases, since the number of blocks reduces at least by a factor of 2 going from one phase to the next. This entire process is repeated $O(\log n)$ time and the final outcome is the minimum of the edit distances computed over these repetitions. The following theorem summarizes the performance of this algorithm.

Theorem 3. *There exists an algorithm that obtains an $O(\beta(n) \log z \sqrt{\ln d})$-approximation factor for edit distance computation to* Dyck(s) *for any $s \geq 2$ in $O(n \log n + \alpha(n))$ time with probability at least $\left(1 - \frac{1}{n} - \frac{1}{d}\right)$, where there exists an algorithm for* StrEdit *running in $\alpha(n)$ time that achieves an approximation factor of $\beta(n)$, and z is the number of blocks.*

The $\sqrt{\log d}$ factor in the approximation can be avoided if we consider iterating $O(n^{\epsilon} \log n)$ times. Since, the best known near-linear time algorithm for StrEdit anyway has $\alpha(n) = n^{1+\epsilon}$ and $\beta = (\log n)^{\frac{1}{\epsilon}}$, we obtain the following theorem.

Theorem 4. *For any $\epsilon > 0$, there exists an algorithm that obtains an $O(\frac{1}{\epsilon} \log z (\log n)^{\frac{1}{\epsilon}})$-approximation factor for edit distance computation to* Dyck(s) *for any $s \geq 2$ in $O(n^{1+\epsilon})$ time with high probability, and z is the number of blocks.*

If instead we apply the string edit distance computation algorithm [23] in Thereom 3 and Thereom 4, we get the corollary

Corollary 5. *(i) There exists an algorithm that obtains an $O(\log z \sqrt{\ln d})$-approximation factor for edit distance computation to* Dyck(s) *for any $s \geq 2$ in $O(n + d^2)$ time with high probability, and also compute the edits.*

(ii) There exists an algorithm that obtains an $O(\frac{1}{\epsilon} \log z)$-approximation factor for edit distance computation to Dyck(s) *for any $s \geq 2$ in $O(n^{1+\epsilon} + d^2 n^{\epsilon})$ time with high probability, and also compute the edits.*

The algorithm and its analysis gives a general framework which can be applied to many other languages beyond Dyck(s). Employing this algorithm one can repair footprints of several memory checking languages such as Stack, Queue, PQ and Deque efficiently. We discuss this connection in Section 5.

2 Analysis of Random-deletion

Here we analyse the performance of **Random-deletion** and prove Theorem 1.

Recall the algorithm. It uses a stack and scans σ left to right. Whenever it encounters an open parenthesis, the algorithm pushes it into the stack. Whenever it encounters a close parenthesis, it *compares* this current symbol with the one at the stack top. If they can be matched, the algorithm always matches them and proceeds to the next symbol. If the stack is empty, it deletes the close parenthesis and again proceeds to the next symbol. Else, the stack is non-empty but the two parentheses cannot be matched. In that case, the algorithm deletes one of the symbol, either the one at the stack top or the current close parenthesis in the string. It tosses an unbiased coin, and independently with probability $\frac{1}{2}$ it chooses which one of them to delete. If there is no more close parenthesis, but stack is non-empty, then it deletes all the open parentheses from the stack.

We consider only deletion as a viable edit operation and under deletion-only model, assume that the optimal algorithm is stack based, and matches well-formed substrings greedily. The following two lemmas whose proofs are in the appendix state that we lose only a factor 2 in the approximation by doing so.

Lemma 6. *For any string $\sigma \in (T \cup \bar{T})^*$, $OPT(\sigma) \leq OPT_d(\sigma) \leq 2OPT(\sigma)$, where $OPT(\sigma)$ is the minimum number of edits: insertions, deletions, substitutions required and $OPT_d(\sigma)$ is the minimum number of deletions required to make σ well-formed.*

Lemma 7. *There exists an optimal algorithm that makes a single scan over the input pushing open parentheses to stack and on observing a close parenthesis, the algorithm compares it with the stack top. If the symbols match, then both are removed from further consideration, otherwise one of the two symbols is deleted.*

From now onward we fix a specific optimal stack based algorithm, and refer that as the optimal algorithm.

Let us initiate time $t = 0$. At every step in *Random-deletion* when we either match two parentheses (current close parenthesis in the string with open parenthesis at the stack top) or delete one of them, we increment time t by 1.

We define two sets A_t and A_t^{OPT} for each time t.

Definition 4. *For every time $t \geq 0$, A_t is defined as all the indices of the symbols that are matched or deleted by* **Random-deletion** *up to and including time t.*

Definition 5. *For every time $t \geq 0$,*

$$A_t^{OPT} = \{i \mid i \in A_t \text{ or } i \text{ is matched by } \textbf{the optimal algorithm} \text{ with some symbol with index in } A_t\}.$$

Clearly at all time $t \geq 0$, $A_t^{OPT} \supseteq A_t$. We now define a correct and wrong move.

Definition 6. *A comparison at time t in the algorithm leading to a deletion is a correct move if $|A_t^{OPT} \setminus A_t| \leq |A_{t-1}^{OPT} \setminus A_{t-1}|$ and is a wrong move if $|A_t^{OPT} \setminus A_t| > |A_{t-1}^{OPT} \setminus A_{t-1}|$.*

Lemma 8. *At any time t, there is always a correct move, and hence* **Random-deletion** *always takes a correct move with probability at least $\frac{1}{2}$.*

Proof. Suppose the algorithm compares an open parenthesis $\sigma[i]$ with a close parenthesis $\sigma[j]$, $i < j$ at time t, and they do not match. If possible, suppose that there is no correct move.

Since *Random-deletion* is stack-based, A_t contains all indices in $[i, j]$. It may also contain intervals of indices $[i_1, j_1], [i_2, j_2], \ldots$ because there can be multiple blocks. It must hold $[1, j-1] \setminus A_t$ does not contain any close parenthesis. Now for both the two possible deletions to be wrong, the optimal algorithm must match $\sigma[i]$ with some $\sigma[j']$, $j' > j$, and also match $\sigma[j]$ with $\sigma[i']$, $i' < i, i' \notin A_t$. But, this is not possible due to the property of well-formedness.

Now consider the case that at time t, the stack is empty and the current symbol in the string is $\sigma[j]$. In that case *Random-deletion* deletes $\sigma[j]$. Clearly, $A_t = [1, j]$. For this to be a wrong move, the optimal algorithm should match $\sigma[j]$ with $\sigma[i]$, $i < 1$ which is not possible. Hence, in this case the move is correct.

Now consider the case that at time t, the input is exhausted and the algorithm considers $\sigma[i]$ from the stack top. In that case *Random-deletion* deletes $\sigma[i]$. Clearly A_t contains indices of all close parenthesis. For this to be a wrong move, the optimal algorithm should match $\sigma[i]$ with $\sigma[j]$, $j \notin A_t$ which is not possible. Hence, in this case the move is correct too. □

Lemma 9. *If at time* t *(up to and including time* t*), the number of indices in* A_t^{OPT} *that the optimal algorithm deletes is* d_t *and the number of correct and wrong moves are respectively* c_t *and* w_t *then* $|A_t^{OPT} \setminus A_t| \leq d_t + w_t - c_t$.

The proof is presented in the appendix. It considers various states of the algorithm at time t, and compares A_t^{OPT} with A_t to obtain the bound. Let S_t denote the string σ at time t after removing all the symbols that were deleted by **Random-deletion** up to and including time t.

Lemma 10. *Consider* d *to be the optimal edit distance to* $\text{DYCK}(s)$*. If at time* t *(up to and including* t*), the number of indices in* A_t^{OPT} *that the optimal algorithm deletes be* d_t *and* $|A_t^{OPT} \setminus A_t| = r_t$*, at most* $r_t + (d - d_t)$ *edits is sufficient to convert* S_t *into a well-balanced string.*

Proof. Since $|A_t^{OPT} \setminus A_t| = r_t$, there exists exactly r_t indices in S_t such that if those indices are deleted, the resultant string is same as what the optimal algorithm obtains after processing the symbols in A_t^{OPT}. For the symbols in remaining $\{1, 2, ..., n\} \setminus A_t^{OPT}$, the optimal algorithm does at most $d - d_t$ edits. Therefore a total of $r_t + (d - d_t)$ edits is sufficient to convert S_t into a well-balanced string. □

Lemma 11. *The edit distance between the final string* S_n *and* σ *is at most* $d + 2w_n$.

Proof. Consider any time $t \geq 0$, if at t, the number of deletions by the optimal algorithm in A_t^{OPT} is d_t, the number of correct moves and wrong moves are respectively c_t and w_t, then we have $|A_t^{OPT} \setminus A_t| \leq d_t + w_t - c_t$. The number of edits that have been performed to get S_t from S_0 is $c_t + w_t$. Denote this by $E(0, t)$. The number of edits that are required to transform S_t to well-formed is at most $(d - d_t) + d_t + w_t - c_t = d + w_t - c_t$ (by Lemma 10). Denote it by $E'(t, n)$. Hence the minimum total number of edits required (including those already performed) considering state at time t is $E(0, t) + E'(t, n) = d + 2w_t$. Since this holds for all time t, the lemma is established. □

In order to bound the edit distance, we need a bound on w_n. To do so we map the process of deletions by **Random-deletion** to a random walk.

2.1 Mapping into Random Walk

We consider the following one dimensional random walk. The random walk starts at coordinate d, at each step, it moves one step right ($+1$) with probability $\frac{1}{2}$ and moves one step left (-1) with probability $\frac{1}{2}$. We count the number of steps required by the random walk to hit the origin.

We now associate a modified random walk with the deletions performed by *Random-deletion* as follows. Every time *Random-deletion* needs to take a move (performs one deletion), we consider one step of the modified random walk. If *Random-deletion* takes a wrong move, we let this random walk make a right (away from origin) step. On the other hand if *Random-deletion* takes a correct move, we let this random walk take a left step (towards origin move). If the random walk takes W right steps, then *Random-deletion* also makes W wrong moves. If the random walk takes W right steps before hitting the origin, then it takes in total a $d + 2W$ steps, and *Random-deletion* also deletes $d + 2W$ times. Therefore, hitting time of this modified random walk starting from d characterizes the number of edit operations performed by *Random-deletion*. In this random walk, left steps (towards origin) are taken with probability $\geq \frac{1}{2}$ (sometimes with probability 1). Therefore, hitting time of this modified random walk is always less than the hitting time of an one-dimensional random walk starting at d and taking right and left step independently with equal probability.

We therefore calculate the probability of a one-dimensional random walk taking right or left steps with equal probability to have a hitting time D starting from d. The computed probability serves as a lower bound on the probability that *Random-deletion* takes D edit operations to transform σ to well-formed.

The one dimensional random walk is related to GAMBLER'S RUIN problem. In gambler's ruin, a gambler enters a casino with $\$d$ and starts playing a game where he wins with probability $1/2$ and loses with probability $1/2$ independently. The gambler plays the game repeatedly betting $\$1$ in each round. He leaves if he runs out of money or his total fortune reaches $\$N$. In gambler's ruin problem one is interested in the hitting probability of the absorbing states. For us, we are interested in the probability that gambler gets ruined. We can set $N = n + d$ because that implies the random walk needs to take n steps at the least to reach fortune and n is a trivial upper bound on the edit distance.

Let $\mathcal{P}_d$ denote the probability that gambler is ruined or his fortune reaches $\$N$ on the condition that his current fortune is $\$d$ and also let $\mathcal{E}_d$ denote the expected number of steps needed for the gambler to be ruined or get $\$N$ starting from $\$d$. Then it is easy to see using Markov property that the distribution $\mathcal{P}_d$ satisfies the following recursion $\mathcal{P}_d = \frac{1}{2}\mathcal{P}_{d+1} + \frac{1}{2}\mathcal{P}_{d-1}$. It follows from the above that the expectation $\mathcal{E}_d$ satisfies $\mathcal{E}_d = \frac{1}{2}\left(\mathcal{E}_{d+1}+1\right) + \frac{1}{2}\left(\mathcal{E}_{d-1}+1\right) = \frac{1}{2}\left(\mathcal{E}_{d+1}+\mathcal{E}_{d-1}\right)+1$. Solving the recursion one gets $\mathcal{E}_d = d(N-d)$, which is useless in our case. But note that even though $\mathcal{E}_1 = N-1$, with probability $\frac{1}{2}$, the gambler is ruined in just 1 step. This indicates lack of concentration around the expectation. Indeed, the distribution $\mathcal{P}_d$ is heavy-tailed which can be exploited to bound the hitting time.

We now calculate the probability that the gambler is ruined in D steps precisely. Let $\mathcal{P}_d$ denote the law of a random walk starting in $d \geq 0$, let $\{Y_i\}_0^\infty$ be the i.i.d. steps of the random walk, let $S_D = d + Y_1 + Y_2 + \ldots + Y_D$ be the position of random walk starting in position d after D steps, and let $T_0 = \inf D : S_D = 0$ denotes the walks first hitting time of the origin. Clearly $T_0 = 0$ for $\mathcal{P}_0$. Then we can show

Lemma 12. *For the* GAMBLER'S RUIN *problem* $\mathcal{P}_d(T_0 = D) = \frac{d}{D}\left(\begin{array}{c} D \\ \frac{D-d}{2} \end{array}\right)\frac{1}{2^D}$.

Proof. We first calculate $\mathcal{P}_d(S_D = 0)$. In order for a random walk to be at position 0 starting at $+d$, there must be $r = (D-d)/2$ indices $i_1, i_2, \ldots, i_r$ such that $Y_{i_1} = Y_{i_2} = \ldots = Y_{i_r} = +1$. Rest of the $\frac{D-d}{2} + d = r + d$ steps must be -1. Hence $\mathcal{P}_d(S_D = 0) = \left(\begin{array}{c} D \\ \frac{D-d}{2} \end{array}\right)\frac{1}{2^D}$, and the lemma follows from the following hitting time theorem.

Theorem (Hitting Time Theorem [34])**.** *For a random walk starting in $D \geq 1$ with i.i.d. steps $\{Y_i\}_0^\infty$ satisfying that $Y_i \geq -1$ almost surely, the distribution of T_0 under $\mathcal{P}_d$ is given by*

$$\mathcal{P}_d\left(T_0 = D\right) = \frac{d}{D}\mathcal{P}_d\left(S_D = 0\right). \tag{1}$$

□

We now calculate the probability that a gambler starting with $\$d$ hits 0 within cd steps. Our goal will be to minimize c as much as possible, yet achieving a significant probability of hitting 0.

Lemma 13. *In* GAMBLER'S RUIN*, the gambler starting with $\$d$ hits 0 within $2d^2$ steps with probability at least 0.194.*

Proof. From Lemma (12) $\mathcal{P}_d(T_0 = cd) = \frac{d}{cd}\left(\begin{array}{c} cd \\ \frac{cd-d}{2} \end{array}\right)\frac{1}{2^{cd}}$. We now employ the following inequality to bound $\left(\begin{array}{c} cd \\ \frac{cd-d}{2} \end{array}\right)\frac{1}{2^{cd}}$.

Lemma (Lemma 7, Ch. 10 [25])**.** *(An estimate for a binomial coefficient.) Suppose λm is an integer where $0 < \lambda < 1$. Then*

$$\frac{1}{\sqrt{8m\lambda(1-\lambda)}}2^{mH_2(\lambda)} \leq \left(\begin{array}{c} m \\ \lambda m \end{array}\right) \leq \frac{1}{\sqrt{2\pi m\lambda(1-\lambda)}}2^{mH_2(\lambda)}$$

where H_2 is the binary entropy function.

We have $\lambda = \frac{1}{2}\left(1-\frac{1}{c}\right)$ and $\binom{cd}{\frac{d(c-1)}{2}} \geq \frac{2^{cdH_2(\frac{1}{2}\left(1-\frac{1}{c}\right))}}{\sqrt{2cd\left(1-\frac{1}{c}\right)\left(1+\frac{1}{c}\right)}} = \frac{2^{cdH_2(\frac{1}{2}\left(1-\frac{1}{c}\right))}}{\sqrt{2cd\left(1-\frac{1}{c^2}\right)}}$.

Therefore, $\mathcal{P}_d(T_0 = cd) = \frac{1}{c}\binom{cd}{\frac{d(c-1)}{2}}\frac{1}{2^{cd}} \geq \frac{1}{c}\frac{2^{cd(H_2(\frac{1}{2}\left(1-\frac{1}{c}\right))-1)}}{\sqrt{2cd\left(1-\frac{1}{c^2}\right)}}$. We now use the Taylor series expansion for $H(x)$ around $\frac{1}{2}$, $1 - H(\frac{1}{2}-x) = \frac{2}{\ln 2}(x^2 + O(x^4))$. Hence,

$$\mathcal{P}_d(T_0 = cd) \geq \frac{1}{c\sqrt{2cd\left(1-\frac{1}{c^2}\right)}}2^{-\frac{2}{\ln 2}cd\frac{1}{4c^2}} = \frac{1}{cd\sqrt{2\frac{c}{d}\left(1-\frac{1}{c^2}\right)}}2^{-\frac{1}{2\frac{c}{d}\ln 2}} > \frac{e^{-\frac{d}{2c}}}{c\sqrt{2cd}} = \frac{1}{c\sqrt{2cd}e^{\frac{d}{2c}}},$$

Now set $c = \alpha d$, $\alpha > 0$ to get $\mathcal{P}_d(T_0 = \alpha d^2) \geq \frac{1}{\alpha d^2\sqrt{2\alpha}e^{\frac{1}{2\alpha}}} = \frac{A(\alpha)}{d^2}$. where $A(\alpha) = \frac{1}{\alpha\sqrt{2\alpha}e^{\frac{1}{2\alpha}}}$. Now $A(\alpha)$ is a decreasing function of $\alpha \geq \frac{1}{3}$ (derivative of $\ln A(\alpha)$ is negative for $\alpha \geq \frac{1}{3}$). Therefore, we have, $\mathcal{P}_d(d^2 \leq T_0 \leq 2d^2) \geq d^2\frac{A(2)}{d^2} = A(2) = 0.194$. □

Corollary 14. *In* GAMBLER'S RUIN, *the gambler starting with* $\$d$ *hits* 0 *within* $\frac{1}{\epsilon}\frac{d^2}{\log d}$ *steps for any constant* $\epsilon > 0$ *with probability at least* $\frac{\sqrt{\epsilon \log d}}{d^\epsilon}$.

Proof. Let $\epsilon' = 2\epsilon$. Set $c = \frac{1}{\epsilon'}\frac{d}{\log d}$ in the above proof, that is $\alpha = \frac{1}{\epsilon' \log d}$ to get $\mathcal{P}_d(T_0 = \frac{1}{\epsilon'}\frac{d^2}{\log d}) \geq \frac{(\epsilon' \log d)^{3/2}}{d^2\sqrt{2}e^{\frac{\epsilon' \log d}{2}}} = \frac{(\epsilon' \log d)^{3/2}}{d^2\sqrt{2}}\frac{1}{d^{\epsilon'/2}}$. Considering $A(\alpha)$ is an increasing function when $\alpha < \frac{1}{3}$, we get

$$\mathcal{P}_d(\frac{1}{\epsilon'}\frac{d^2}{\log d} \leq T_0 \leq \frac{2}{\epsilon'}\frac{d^2}{\log d}) \geq \sqrt{\frac{\epsilon' \log d}{2}}\frac{1}{d^{\epsilon'/2}}.$$

Now putting $\epsilon = \frac{\epsilon'}{2}$, we get the result. □

Theorem 15. Random-deletion *obtains a* $2d$*-approximation for* DYCK(s) *language edit distance problem for any* $s \geq 2$ *in linear time with constant probability, when only deletions are allowed.*

Proof. The theorem follows from the mapping that the edit distance computed by *Random-deletion* is at most the number of steps taken by gambler's ruin to hit the origin starting from $\$d$ and then applying Lemma 13. □

Theorem (1)**. Random-deletion** *obtains a* $4d$*-approximation for edit distance computations (substitution, insertion, deletion) to* DYCK(s) *for any* $s \geq 2$ *in linear time with constant probability.*

Proof. Follows from the previous theorem, Lemma 6 and Lemma 7. □

3 Analysis of the Refined Algorithm

We revisit the description of the refined algorithm. Given a string σ, we first remove any prefix of close parentheses and any suffix of open parentheses to start with. Since any optimal algorithm will also remove them, this does not affect the edit distance. We also remove any well-formed substrings since that does not affect the edit distance as well (Lemma 7). Now one can write $\sigma = Y_1X_1Y_2X_2...Y_zX_z$ where each $Y_i, i = 1,2,..,z$ consists of only open parentheses and each $X_i, i = 1,2,..,z$ consists of only close parentheses. We call each Y_iX_i a *block*. After well-formed substrings removal, each block requires at least one edit. Hence, we have $z \leq d$ where d is the optimal edit distance of string σ.

We can write σ based on the processing of the optimal algorithm as follows

$$\sigma = Z_{1,z}Z_{1,z-1}...Z_{1,1}X_1Z_{2,z}Z_{2,z-1}...Z_{2,3}Z_{2,2}X_2......Z_{z-1,z}Z_{z-1,z-1}X_{z-1}Z_{z,z}X_z.$$

Here the optimal algorithm matches X_1 with $Z_{1,1}$, that is the close parentheses of X_1 are only matched with open parentheses in $Z_{1,1}$ and vice-versa. Some parentheses in X_1 and $Z_{1,1}$ may need to be deleted for matching X_1 with $Z_{1,1}$ using minimum number of edits. Similarly, the optimal algorithm matches X_2 with $Z_{2,1}Z_{2,2}$, matches X_3 with $Z_{1,3}Z_{2,3}Z_{3,3}$ and so on. Note that it is possible that some of these $Z_{i,j}$ $i = 1, 2, .., z, j \geq i$ may be empty.

The algorithm proceeds as follows. It continues **Random-deletion** process as before, but now it keeps track of the substring with which each X_a, $a = 1, 2, .., z$ is matched (possibly through multiple deletions) during this random deletion process. While processing X_1, the random deletion process is restarted $3 \log_b n$ times, $b = \frac{1}{(1-0.194)} = 1.24$ and at each time the algorithm keeps a count on how many deletions are necessary to complete processing of X_1. It then selects the particular iteration in which the number of deletions is minimum. We let $Z_{1,min}$ to denote the substring to which X_1 is matched in that iteration. The algorithm then continues the random deletion process. It next stops when processing on X_2 finishes. Again, the portion of random deletion process between completion of processing X_1 and completion of processing X_2 is repeated $3 \log_b n$ times and the iteration that results in minimum number of deletions is picked. We define $Z_{2,min}$ accordingly. The algorithm keeps proceeding in a similar manner until the string is exhausted. In the process, $Z_{a,min}$ is matched with X_a for $a = 1, 2, .., z$. But, instead of using the edits that the random deletion process makes to match $Z_{a,min}$ to X_a, our algorithm invokes the best string edit distance algorithm $StrEdit(Z_{a,min}, X_a)$ which converts $Z_{a,min}$ to R_a and X_a to T_a such that R_aT_a is well-formed. Clearly, at the end we have a well-formed string. The pseudocode of the refined algorithm is given in the appendix (Algorithm 8.1).

3.1 Analysis

We first analyze its running time.

Lemma 16. *The expected running time of Algorithm 8.1 is $O(n \log n + \alpha(n))$ where $\alpha(n)$ is the running time of* StrEdit *to approximate string edit distance of input string of length n within factor $\beta(n)$.*

Proof. First $StrEdit(Z_{a,min}, X_a)$, $a = 1, 2, .., z$ is invoked on disjoint subset of substrings of σ. Let us denote the length of these substrings by $n_1, n_2, .., n_z$. Since the running time $\alpha(n)$ is convex, the total time required to run $StrEdit(Z_{a,min}, X_a)$, $a = 1, 2, .., z$ is

$$\alpha(n_1) + \alpha(n_2) + ... + \alpha(n_z) \leq \alpha(n_1 + n_2 + ... + n_z) = \alpha(n).$$

We now calculate the total running time of Algorithm 8.1 without considering the running time for StrEdit. While processing X_a, in each of $3 \log_b n$ rounds, the running time is bounded by number of comparisons that our algorithm makes with symbols in X_a. Each comparison might result in matching of a symbol in X_a–in that case it adds one to the running time, deletion of the symbol in X_a, again it adds one to the running time, and deletion of open parenthesis while comparing with a symbol in X_a–this might require multiple comparisons with the same symbol of X_a. If there are η such comparisons with some symbol $x \in X_a$, then there are $\eta - 1$ consecutive comparisons with x, where *random-deletion* chooses to delete the open parenthesis. On the ηth comparison, x is either matched or deleted. Now the probability that *random-deletion* deletes $(\eta - 1)$ open parenthesis while comparing with x is $\frac{1}{2^{\eta-1}}$. Therefore, the expected number of comparisons involving each symbol $x \in X_a$ before it gets deleted or matched is $\sum_{i \geq 1} \frac{i}{2^i} \leq 2$. Hence the expected number of total comparisons in each iteration involving X_a is at most $3|X_a|$. Therefore, the total expected number of comparison over all the iterations and $a = 1, 2, .., z$ is at most $\sum_a 9|X_a| \log_b n$.

Of course, if the running time becomes more than say 2 times the expected value, we can restart the entire process. By Markov inequality on expectation 2 rounds is sufficient to ensure that the algorithm will make at most $\sum_a 18|X_a|\log_b n$ comparisons in at least one round. Hence the lemma follows. □

We now proceed to analyze the approximation factor. For that we assume that the optimal distance d is at least 3 (otherwise employ the algorithm of the previous section). Let the optimal edit distance to convert $Z_a X_a$ into well-formed be d_a for $a = 1, 2, .., z$ where $Z_a = Z_{1,a} Z_{2,a} ... Z_{a,a}$.

While computing the set $Z_{a,min}$, it is possible that our algorithm inserts symbols outside of Z_a to it or leaves out some symbols of Z_a. *In the former case, among the extra symbols that are added, if the optimal algorithm deletes some of these symbols as part of some other $Z_{a'}$, $a' \neq a$, then these deletions are "harmless". If we only include these extra symbols to $Z_{a,min}$, then we can as well pretend that those symbols are included in Z_a too. The edit distance of the remaining substrings are not affected by this modification. Therefore, for analysis purpose,* **both for this algorithm and for the main algorithm in the next section**, *we always assume w.l.o.g that the optimal algorithm does not delete any of the extra symbols that are added.*

Lemma 17. *The number of deletions made by random deletion process to finish processing $X_1, X_2, ..X_l$, for $l = 1, 2, .., z$, that is to match $Z_{a,min}, X_a$, $a \leq l$, is at most $2(\sum_{a=1}^{l} d_a)^2$ with probability at least $\left(1 - \frac{1}{n^3}\right)^l$.*

Proof. Consider $l = 1$, that is only X_1. The number of deletions made by random deletion process to finish processing X_1 is at most the hitting time of an one dimensional random walk starting from position d_1. This follows from Section 2.1 and noting that any deletion outside of Z_1 is a wrong deletion. From Lemma 13, the hitting time of a random walk starting from d_1 is at most $2d_1^2$ with probability at least 0.194. Let $b = \frac{1}{(1-0.194)} = 1.24$. Since, we repeat the process $3\log_b n$ times, the probability that the minimum hitting time among these $3\log_b n$ iterations is more than $2d_1^2$ is at most $(1 - 0.194)^{3\log_b n} = \frac{1}{n^3}$. Therefore, the number of deletions made by random deletion process to finish processing X_1 is at most $2d_1^2$ with probability $\left(1 - \frac{1}{n^3}\right)$.

Now consider $l = 2$, the number of deletions made by random deletion process to finish processing X_1, X_2 is at most the hitting time of an one dimensional random walk starting from position $d_1 + d_2$. Start the random walk from $(d_1 + d_2)$ and first follow the steps as suggested by $Z_{1,min}$ to hit d_2 in at most $2d_1^2$ steps. Now, since we repeat the random deletion process from the time of completing processing of X_1 to completing processing of X_2, then by similar argument as in $l = 1$, the minimum hitting time starting from d_2 is at most $2d_2^2$ with probability $\left(1 - \frac{1}{n^3}\right)$. Hence, the minimum hitting time starting from $d_1 + d_2$ is at most $2d_1^2 + 2d_2^2 < 2(d_1 + d_2)^2$ with probability $\left(1 - \frac{1}{n^3}\right)^2$.

Proceeding in a similar manner for $l = 3, .., z$, we get the lemma. □

Let us denote by C_l and W_l the number of correct and wrong moves taken by the random deletion process when processing of $X_1, X_2, ..., X_l$ finishes. Since at each deletion, correct move has been taken with probability at least $\frac{1}{2}$ then by standard Chernoff bound followed by union bound we have the following lemma.

Lemma 18. *When the processing of $X_1, X_2, ..., X_l$ finishes $W_l - C_l \leq (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d}$ with probability at least $\left(1 - \frac{1}{n} - \frac{1}{d}\right)$.*

Proof. Probability that the number of deletions made by random deletion process is at most $2(\sum_{a=1}^{l} d_a)^2$ is $\geq \left(1 - \frac{1}{n^3}\right)^l$ by Lemma 17. Let us denote the number of these deletions by D_l, for $l = 1, 2, ..., z$. Hence $D_l = W_l + C_l$. We have $\mathsf{E}[C_l] \geq \frac{D_l}{2}$ and $\mathsf{E}[W_l] \leq \frac{D_l}{2}$. We use the following version of Azuma's inequality for simple coin flips to bound $W_l - C_l$.

Azuma's inequality for coin flips. *Let F_i be a sequence of independent and identically distributed random coin flips (i.e., let F_i be equally likely to be -1 or 1 independent of the other values of F_i). Defining $X_i = \sum_{j=1}^{i} F_j$ yields a martingale with $|X_k X_{k1}|1$, allowing us to apply generic Azuma's inequality to get*

$$\Pr\left[X_N > t\right] \leq \exp\left(\frac{-t^2}{2N}\right).$$

Note that for our case, probability of a wrong move is at most the probability of a correct move, but we are only interested to bound $W_l - C_l$ and not the absolute difference. This difference will be maximum when wrong and correct moves are taken with equal probability. Hence, we can apply the above Azuma's inequality.

$$\Pr\left[W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d} | D_1 \leq 2d_1^2\right] \leq exp^{-}\left(\frac{8(\sum_{a=1}^{l} d_a)^2 \ln d}{4(\sum_{a=1}^{l} d_a)^2}\right) = \frac{1}{d^2}.$$

Hence

$$\begin{aligned}
& \Pr\left[W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d}\right] \\
\leq\; & \Pr\left[W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d} | D_1 > 2d_1^2\right] \Pr\left[D_1 > 2d_1^2\right] \\
& + \Pr\left[W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d} | D_1 \leq 2d_1^2\right] \Pr\left[D_1 \leq 2d_1^2\right] \\
\leq\; & \Pr\left[D_1 > 2d_1^2\right] + \Pr\left[W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d} | D_1 \leq 2d_1^2\right] \Pr\left[D_1 \leq 2d_1^2\right] \\
\leq\; & 1 - \left(1 - \frac{1}{n^3}\right)^l + \left(1 - \frac{1}{n^3}\right)^l \frac{1}{d^2} \leq \frac{l}{n^3} + \frac{1}{d^2}.
\end{aligned}$$

Hence

$$\Pr\left[\exists l \in [1, z] s.t. W_l - C_l > (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d}\right] \leq \sum_{l=1}^{z}\left(\frac{l}{n^3} + \frac{1}{d^2}\right) \leq \frac{z^2}{n^3} + \frac{z}{d^2} \leq \frac{1}{n} + \frac{1}{d}.$$

□

Now we define A_l^{OPT} and A_l in a similar manner as in the previous section. We only consider the iterations that correspond to computing $Z_{a,min}$, $a = 1, 2, .., z$ to define the final random deletion process.

Definition 7. *A_l is defined as all the indices of the symbols that are matched or deleted by* **Random-deletion** *process up to and including time when processing of X_l finishes.*

Definition 8. *For every time $l \in [1, z]$,*

$A_l^{OPT} = \{i \mid i \in A_l$ *or i is matched by* **the optimal algorithm** *with some symbol with index in $A_l\}$.*

We have the following corollary

Corollary 19. *For all $l \in [1, z]$, $|A_l^{OPT} \setminus A_l| \leq \sum_{a=1}^{l} d_a + (2\sum_{a=1}^{l} d_a)\sqrt{2\ln d}$ with probability at least $\left(1 - \frac{1}{n} - \frac{1}{d}\right)$.*

Proof. Proof follows from Lemma 9 and Lemma 18. ☐

Lemma 20. *For all* $a \in [1, z]$, $StrEdit(Z_{a,min}, X_a) \leq d_a + |A_{a-1}^{OPT} \setminus A_{a-1}| + |A_a^{OPT} \setminus A_a|$.

Proof. Let $D(X_a)$ denote the symbols from X_a for which the matching open parentheses have already been deleted before processing on X_a started. Let $D'(X_a)$ denote the symbols from X_a for which the matching open parentheses are not included in $Z_{a,min}$. Let $E(Z_{a,min})$ denote open parentheses in $Z_{a,min}$ such that their matching close parentheses are in X'_a, $a' < a$, that is they are already deleted. Let $E'(Z_{a,min})$ denote open parentheses in $Z_{a,min}$ such that their matching close parentheses are in X'_a, $a' > a$, that is they are extra symbols from higher blocks.

$$StrEdit(Z_{a,min}, X_a) = StrEdit(Z_a, X_a) + |D(X_a)| + |D'(X_a)| + |E(Z_{a,min})| + |E'(Z_{a,min})|.$$

Now all the indices of $D(X_a) \cup E(Z_{a,min})$ are in A_{a-1}^{OPT}, but none of them are in A_{a-1}. Hence

$$|D(X_a)| + |E(Z_{a,min})| \leq |A_{a-1}^{OPT} \setminus A_{a-1}|.$$

Also, the indices corresponding to matching close parenthesis of $E'(Z_{a,min})$ and $D'(X_a)$ are in A_a^{OPT} but not in A_a. Hence

$$|D'(X_a)| + |E'(Z_{a,min})| \leq |A_a^{OPT} \setminus A_a|.$$

Therefore, the lemma follows. ☐

In fact, we can have a stronger version of the above lemma, though it does not help in obtaining a better bound for Theorem 2.

Lemma 21. *For all* $a \in [1, z]$, $StrEdit(Z_{a,min}, X_a) \leq d_a + |A_{a-1}^{OPT} \setminus A_{a-1}| + |\{A_a^{OPT} \setminus A_a\} \setminus \{A_{a-1}^{OPT} \setminus A_{a-1}\}|$.

Proof. Follows from Lemma 20 and noting that

$$|D'(X_a)| + |E'(Z_{a,min})| \leq |\{A_a^{OPT} \setminus A_a\} \setminus \{A_{a-1}^{OPT} \setminus A_{a-1}\}|.$$

☐

Theorem 22 (2)**.** *Algorithm 8.1 obtains an* $O(z\beta(n)\sqrt{\ln d})$*-approximation factor for edit distance computation to* DYCK(s) *for any* $s \geq 2$ *in* $O(n \log n + \alpha(n))$ *time with probability at least* $\left(1 - \frac{1}{n} - \frac{1}{d}\right)$*, where there exists an algorithm for* STREDIT *running in* $\alpha(n)$ *time achieves an approximation factor of* $\beta(n)$.

Proof. The edit distance computed by the refined algorithm is at most $\sum_{a=1}^{z} StrEdit(Z_{a,min}, X_a)$, where recall z is the number of blocks. Hence by Lemma 20 the computed edit distance (assuming we use optimal algorithm for STREDIT) is at most

$$\sum_{a=1}^{z} d_a + |A_{a-1}^{OPT} \setminus A_{a-1}| + |A_a^{OPT} \setminus A_a| = d + 2\sum_{a=1}^{z} |A_a^{OPT} \setminus A_a|$$

$$\leq 7zd\sqrt{2\ln d}, \text{ by Corollary 19.}$$

Since, we use a $\beta(n)$-approximate algorithm for STREDIT and that runs in $\alpha(n)$ time, we get an $O(\beta(n)z\sqrt{\ln d})$-approximation with running time $O(n \log n + \alpha(n))$ (Lemma 16). Hence the theorem follows. ☐

Note. Instead of repeating different portions of random deletion process and then stitching the random deletions corresponding to $Z_{a,min}, a = 1, 2, .., z$, we can simply repeat the entire *Random-deletion* $\Theta(\log n)$ time. For each repetition, run the entire algorithm and finally return the edit distance corresponding to the iteration that returns the minimum value. We get the same approximation factor and asymptotic running time by doing this. However, we need to calculate STREDIT in each iterations, which is not the case in the described algorithm. Therefore, in practice, we save some running time.

4 Further Refinement: Main Algorithm & Analysis

We again view our input σ as $Y_1X_1Y_2X_2...Y_zX_z$, where each Y_a is a sequence of open parentheses and each X_a is a sequence of close parentheses. Each Y_a, X_a, $a = 1, 2, .., z$, is a block. We know by preprocessing, $z \leq d$, where d is the optimal edit distance to $\text{DYCK}(s)$.

As before, we first run the process of **Random-deletion**. For each run of *Random-deletion*, the algorithm proceeds in phases with at most $\lceil \log_2 z \rceil + 1$ phases. We repeat this entire procedure $3 \log_b n$ times, $b = 1.24$ (as before) and return the minimum edit distance computed over these runs and the corresponding well-formed string. We now describe the algorithm corresponding to a single run of *random-deletion* (also shown pictorially in Figure 1).

Let us use $X_a^1 = X_a, Y_a^1 = Y_a$ to denote the blocks in the first phase. Consider the part of *Random-deletion* from the start of processing X_a^1 to finish either X_a^1 or Y_a^1 whichever happens first. Since this part of the random deletion (from the start of X_a^1 to the completion of either X_a^1 or Y_a^1) is contained entirely within block $Y_a^1X_a^1$, we call this part $local^1$ *to block* a. Let $A_a^{local^1}$ denote the indices of all the symbols that are matched or deleted during the $local^1$ steps in block a. Let $A_a^{OPT,local^1}$ be the union of $A_a^{local^1}$ and the indices of symbols that are matched with some symbol with indices in $A_a^{local^1}$ in the optimal solution. We call $A_a^{OPT,local^1} \setminus A_a^{local^1}$ the $local^1$ error, denoted $local\text{-}error^1(Y_a^1, X_a^1)$.

Create substrings L_a^1 corresponding to $local^1$ moves in block a, $a = 1, .., z$. Compute STREDIT between $L_a^1 \cap Y_a^1$ to $L_a^1 \cap X_a^1$. Remove all these substrings from further consideration. The phase 1 ends here.

We can now view the remaining string as $Y_1^2X_1^2Y_2^2X_2^2...Y_{z^2}^2X_{z^2}^2$, after deleting any prefix of open parentheses and any suffix of close parentheses. Consider any Y_a^2, X_a^2. Let they span the original blocks $Y_{a_1}X_{a_1}Y_{a_1+1}X_{a_1+1}...Y_{a_2}X_{a_2}$. Consider the part of *Random-deletion* from the start of processing X_{a_1} to the completion of either Y_{a_1} or X_{a_2} whichever happens first. Since this part of the random deletion remains confined within block $Y_a^2X_a^2$, we call this part $local^2$ *to block* a. Let $A_a^{local^2}$ denote the indices of all the symbols that are matched or deleted during the $local^2$ steps in block a. Let $A_a^{OPT,local^2}$ be the union of $A_a^{local^2}$ and the indices of symbols that are matched with some symbol with indices in $A_a^{local^2}$ in the optimal solution. We call $A_a^{OPT,local^2} \setminus A_a^{local^2}$ the $local^2$ error, denoted $local\text{-}error^2(Y_a^2, X_a^2)$.

Create substrings L_a^2 corresponding to $local^2$ moves in block a, $a = 1, .., z^2$. Compute STREDIT between $L_a^2 \cap Y_a^2$ to $L_a^2 \cap X_a^2$. Remove all these substrings from further consideration.

We continue in this fashion until the remaining string becomes empty. We can define $local^i$ moves, $A_a^{local^i}$, $A_a^{OPT,local^i}$ and $local\text{-}error^i(Y_a^i, X_a^i)$ accordingly.

Definition 9. *For ith phase blocks $Y_a^iX_a^i$, if they span original blocks $Y_{a_1}X_{a_1}Y_{a_1+1}X_{a_1+1}...Y_{a_2}X_{a_2}$, then part of random deletion from the start of processing X_{a_1} to finish either Y_{a_1} or X_{a_2} whichever happens first, remains confined in block $Y_a^iX_a^i$ and is defined as $local^i$ move.*

Definition 10. *For any $i \in \mathbb{N}$, $A_a^{local^i}$ denote the indices of all the symbols that are matched or deleted during the $local^i$ steps in block a.*

Definition 11. *For any $i \in \mathbb{N}$, $A_a^{OPT,local^i}$ denote the union of $A_a^{local^i}$ and the indices of symbols matched with some symbol in $A_a^{local^i}$.*

Definition 12. *For any $i \in \mathbb{N}$, $A_a^{OPT,local^i} \setminus A_a^{local^i}$ is defined as the $local^i$ error, $local\text{-}error^i$.*

We now summarize the algorithm below.

Algorithm:

Given the input $\sigma = Y_1X_1Y_2X_2...Y_zX_z$, the algorithm is as follows

- $MinEdit = \infty$,

- For $iteration = 1, iteration \leq 3\log_b n, iteration++$
  - Run **Random-deletion** process.
  - Set $i = 1, z^1 = z$, $edit = 0$, and for $a = 1, 2, ..., z^1$, $X_a^1 = X_a, Y_a^1 = Y_a$.
  - While σ is not empty
    * Consider the part of random-deletion from the start of processing X_a^i to finish either X_a^i or Y_a^i whichever happens first.
    * Create substrings L_a^i, $a = 1, 2, .., z^i$ which correspond to $local^i$ moves. Compute $StrEdit(L_a^i \cap Y_a^i, L_a^i \cap X_a^i)$ to match $L_a^i \cap Y_a^i$ to $L_a^i \cap X_a^i$ and add the required number of edits to $edit$.
    * Remove L_a^i, $a = 1, 2, .., z^i$ from σ, write the remaining string as $Y_1^{i+1}X_1^{i+1}Y_2^{i+1}X_2^{i+1}...Y_{z^{i+1}}^{i+1}X_{z^{i+1}}^{i+1}$, possibly by deleting any prefix of close parentheses and any suffix of open parentheses. The number of such deletions is also added to $edit$. Set $i = i + 1$
  - End While
  - If ($edit < MinEdit$) set $MinEdit = edit$
  - End For
- Return $MinEdit$ as the computed edit distance.

Of course, the algorithm can compute the well-formed string by editing the parentheses that have been modified in the process through STREDIT operations.

Lemma 23. *There exists at least one iteration among $3\log_b n$, such that for all $a' \leq b'$, (P1) the number of deletions made by random deletion process between the start of processing $X_{a'}$ and finishing either $X_{b'}$ or $Y_{a'}$ whichever happens first is at most $2d(a', b')^2$, where $d(a', b')$ is the number of deletions the optimal algorithm performs starting from the beginning of $X_{a'}$ to complete either $X_{b'}$ or $Y_{a'}$ whichever happens first with probability at least $\left(1 - \frac{1}{n}\right)$.*

Proof. Consider the random source $\mathcal{S}$ that supplies the random coins for deciding among the two choices of deletions to execute, and consider the outcomes of the random source for each of the $3\log_b n$ iterations of *Random-deletion*.

Now consider any $a', b', a \leq b'$. Suppose that the optimal algorithm finishes $Y_{a'}$ first (the argument when the optimal algorithm finishes $X_{b'}$ first is identical), and let the processing on $Y_{a'}$ completes while comparing with $X_{b''}$, $a' \leq b'' \leq b'$. Consider this portion of the substring, and let the number of deletions performed by the optimal algorithm in this substring be γ. Then the number of deletions made by the random deletion process to finish processing $Y_{a'}$ is at most the hitting time of an one dimensional random walk starting from position γ. From Lemma 13, the hitting time of a random walk starting from γ is at most $2\gamma^2$ with probability at least 0.194. Let $b = \frac{1}{(1-0.194)} = 1.24$. If, we repeat the process $3\log_b n$ times, then the probability that there exists at least one iteration with hitting time not more than $2\gamma^2$ is at least $1 - (1 - 0.194)^{3\log_b n} = 1 - \frac{1}{n^3}$. Now use the outcomes of the random source that have been applied for repeating the entire random walk corresponding to the portion of the random walk starting from $X_{a'}$ and finishing either $X_{b'}$ or $Y_{a'}$ whichever happens first. This has the same effect as repeating the process only for the portion of the random walk.

There are at most $z + (z-1) + (z-2) + ... + 1 < z^2$ possible values of a', b'. Hence the probability that there exists at least one iteration in which (P1) holds for all a', b' is by union bound at least $1 - \frac{z^2}{n^3} = \left(1 - \frac{1}{n}\right)$. □

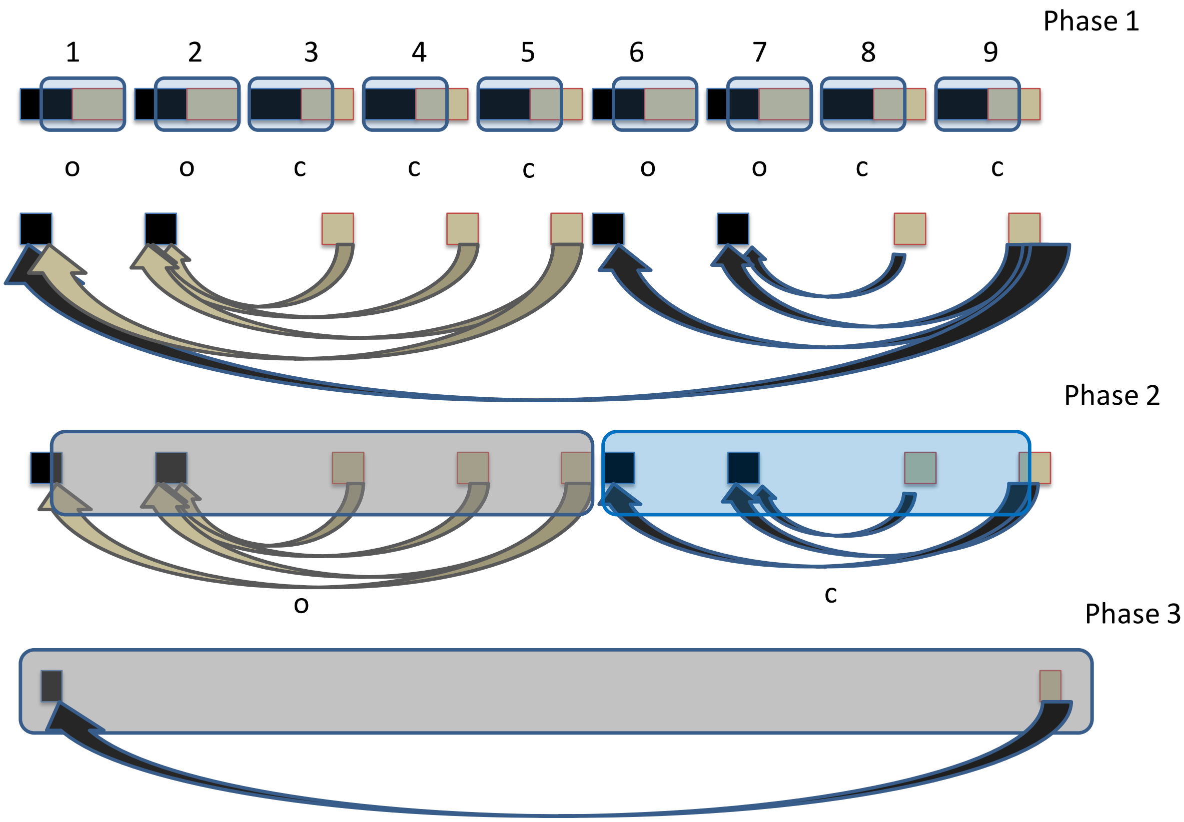

Figure 1: Phase by phase execution of the main algorithm

4.1 Analysis of Approximation Factor

Lemma 24. *Considering phase l which has z^l blocks, the total edit cost paid in phase l of the returned solution is* $\sum_{a=1}^{z^l} StrEdit(L_a^l \cap Y_a^l, L_a^l \cap X_a^l) \le \beta(n)(d+l*d\sqrt{24\ln d})$ *with probability at least* $1-\frac{1}{n}-\frac{1}{d}$.

Proof. Consider the iteration in which Lemma 23 holds, that is we have property (P1). We again fix an optimal stack based algorithm and refer to it as the optimal algorithm.

Phase 1.

Consider Y_a^1, X_a^1. We know $Y_a^1 = Y_a$ and $X_a^1 = X_a$.

- If no symbols of either of Y_a or X_a is matched by the optimal algorithm outside of block a (that is they are matched to each other), then let d_a^1 denote the optimal edit distance to match Y_a with X_a. Consider $d_a^1 \ge 2$.

  By Lemma 23, the random deletion process takes at most $2(d_a^1)^2$ steps within $local_a^1$ moves. Now from Lemma 9, local error, $A_a^{OPT,local^1} \setminus A_a^{local^1} \le d_a^1 + W_a^{local^1} - C_a^{local^1}$ where $W_a^{local^1}$ is the number of wrong moves taken during $local^1$ steps in block a and similarly $C_a^{local^1}$ is the number of correct steps taken during the $local^1$ steps in block a. Since the total number of local steps is at most $2(d_a^1)^2$ and wrong steps are taken with probability at most $\frac{1}{2}$, hence by standard application of Chernoff bound or by Azuma's inequality for simple coin flips as in Lemma 18, local error, $local\text{-}error^1(Y_a^1, X_a^1) \leq d_a^1 + d_a^1\sqrt{12\ln d_a^1} \leq d_a^1\sqrt{24\ln d_a^1}$ with probability at least $1 - \frac{1}{d^3}$.

Hence $StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq d_a + local\text{-}error^1(Y_a^1, X_a^1) \leq d_a^1 + d_a^1\sqrt{24\log d_a^1}$. If $d_a^1 = 1$, then $local\text{-}error^1(Y_a^1, X_a^1) \leq 2$ and $StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq 3d_a$.

- If some suffix of X_a^1 is matched outside of block a, then let $X_a^{1,p}$ be the prefix of X_a^1 which is matched inside. Consider $Y_a^1 X_a^{1,p}$. Let d_a^1 denote the optimal edit distance to match Y_a with X_a^p. Now again, the total number of local steps is at most $2(d_a^1)^2$ and wrong steps are taken with probability at most $\frac{1}{2}$. Hence we have, local error, $local\text{-}error^1(Y_a^1, X_a^1) \leq d_a^1 + d_a^1\sqrt{12\ln d_a^1} \leq d_a^1\sqrt{24\ln d_a^1}$ with probability at least $1 - \frac{1}{d^3}$.

  Hence again $StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq d_a + local\text{-}error^1(Y_a^1, X_a^1) \leq d_a^1 + d_a^1\sqrt{24\ln d_a^1}$.

- If some prefix of Y_a^1 is matched outside of block a, then let $Y_a^{1,s}$ be the suffix of Y_a^1 which is matched inside. Consider $Y_a^{1,a} X_a^1$ and let d_a^1 denote the optimal edit distance to match $Y_a^{1,s}$ with X_a. Again similar arguments lead to $StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq d_a^1 + local\text{-}error^1(Y_a^1, X_a^1) \leq d_a^1 + d_a^1\sqrt{24\ln d_a^1}$.

Hence due to phase 1, the total edit cost paid is at most $\sum_{a=1}^{z} StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq \sum_{a=1}^{z} d_a^1 + d_a^1\sqrt{24\ln d_a^1} \leq d + d\sqrt{24\ln d}$. This also holds when d_a^1 or d is small.

Phase 2.

Consider Y_a^2, X_a^2. Let Y_a^2, X_a^2 spans original blocks $Y_g X_g Y_{g+1} X_{g+1} \ldots, Y_h X_h$ (we drop the superscript 1 for the original blocks).

- If both X_a^2 and Y_a^2 are matched by the optimal algorithm inside g to h blocks, then let d_g^h denote the optimal edit distance to match $Y_g X_g Y_{g+1} X_{g+1} \ldots, Y_h X_h$ and $d_g^h \geq 2$. $d_g^h = 1$ is a trivial case and can be handled as in phase 1.

  By Lemma 23, the random deletion process takes at most $2(d_g^h)^2$ steps within blocks g to h. Hence $local\text{-}error^2(Y_a^2, X_a^2) = A_a^{OPT,local^2} \setminus A_a^{local^2} \leq d_g^h + d_g^h\sqrt{12\ln d_g^h} \leq d_g^h\sqrt{24\ln d_g^h}$. Suppose, *Random-deletion* selects $R \subset Y_a^2$ and $T \subset X_a^2$ to match. Note that either $R = Y_a^2$ or $T = X_a^2$. Let $D(R,T)$ denote all the symbols in R, T such that their matching parentheses belong to blocks either outside of index $[g,h]$ or they exist at phase 2 but are not included. Let $E(R,T)$ denote all the symbols in R, T such that their matching parentheses belong to blocks $[g,h]$ but have already been deleted in phase 1.

  Then
  $$StrEdit(L_a^2 \cap Y_a^2, L_a^2 \cap Y_a^2) \leq d_g^h + |D(R,T)| + |E(R,T)|$$

  Now,
  $$|D(R,T)| \leq local\text{-}error^2(Y_a^2, X_a^2) = A_a^{OPT,local^2} \setminus A_a^{local^2} \leq d_g^h\sqrt{24\ln d_g^h}$$

  For each $x \in E(R,T)$ either its matching parentheses y belongs to the same block in phase 1, say $a_i \in [g,h]$, or in different blocks, say a_i and $a_{i'} \in [g,h]$. In the first case, x can be charged to some y (with which it matches) which contributes 1 to $local\text{-}error^1(Y_{a_i}^1, X_{a_i}^1)$. In the second case, x can again be charged to some y which contributes 1 to $local\text{-}error^1(Y_{a_{i'}}^1, X_{a_{i'}}^1)$. Hence,

  $$|E(R,T)| \leq \sum_{i=g}^{h} local\text{-}error^1(Y_i^1, X_i^1).$$

Thus, we get

$$StrEdit(L_a^2 \cap Y_a^2, L_a^2 \cap Y_a^2) \leq d_g^h + local\text{-}error^2(Y_a^2, X_a^2) + \sum_{i=g}^{h} local\text{-}error^1(Y_i^1, X_i^1)$$

$$\leq d_g^h + d_g^h\sqrt{24 \ln d_g^h} + d_g^h\sqrt{24 \ln d_g^h}$$

$$\leq d_g^h + 2d_g^h\sqrt{24 \ln d_g^h}$$

- If some symbol of X_a^2 is matched outside, let us denote the set of those indices by Υ. Consider the subsequence of X_a^2 excluding Υ. Let us denote it by $X_a^{2,\bar{\Upsilon}}$. Consider $Y_a^2 X_a^{2,\bar{\Upsilon}}$, and proceed as in the previous case.
- If some symbol of Y_a^2 is matched outside, let us denote the set of those indices by Υ. Consider the subsequence of Y_a^2 excluding Υ. Let us denote it by $Y_a^{2,\bar{\Upsilon}}$. Consider $Y_a^{2,\bar{\Upsilon}} X_a^2$, and proceed as in the previous case.

Hence due to phase 2, the total edit cost paid is at most $\sum_{a=1}^{z^2} StrEdit(L_a^1 \cap Y_a^1, L_a^1 \cap X_a^1) \leq d + 2d\sqrt{24 \ln d}$.

Phase l.

Proceeding exactly as in Phase 2, let Y_a^l and X_a^l contain blocks from index $[g^{l-1}, h^{l-1}]$ of level $l-1$, all of which together contain blocks $[g^{l-2}, h^{l-2}]$ from level $l-2$ and so on, finally $[g^1, h^1] = [g, h]$ of blocks from level 1. Let d_g^h denote the optimal edit distance to match Y_a^l and X_a^l excluding the symbols that are matched outside of blocks $[g, h]$ by the optimal algorithm, and again assume $d_g^h \geq 2$. $d_g^h = 1$ is easy and can be handled exactly as in phase 1 when d_a^1 was 1.

Suppose, *Random-deletion* selects $R \subset Y_a^l$ and $T \subset X_a^l$ to match. Note that either $R = Y_a^l$ or $T = X_a^l$. Let $D(R, T)$ denote all the symbols in R, T such that their matching parentheses belong to blocks either outside of index $[g, h]$ or they exist at phase l but are not included. Let $E(R, T)$ denote all the symbols in R, T such that their matching parentheses belong to blocks $[g, h]$ but have already been deleted in phases $1, 2, .., l-1$.

Then

$$StrEdit(L_a^l \cap Y_a^l, L_a^l \cap Y_a^l) \leq d_g^h + |D(R, T)| + |E(R, T)|$$

Now,

$$|D(R, T)| \leq local\text{-}error^l(Y_a^l, X_a^l) = A_a^{OPT, local^l} \setminus A_a^{local^l} \leq d_g^h + d_g^h\sqrt{12 \ln d_g^h} \leq d_g^h\sqrt{20 \ln d_g^h}.$$

For each $x \in E(R, T)$, consider the largest phase $\eta \in [1, 2, .., l-1]$ such that its matching parenthesis y existed before start of the ηth phase but does not exist after the end of the ηth phase. It is possible, either y belongs to the same block in phase η, say $r \in [g^\eta, h^\eta]$, or in different blocks, say r and $s \in [g^\eta, h^\eta]$. In the first case, x can be charged to y which contributes 1 to $local\text{-}error^\eta(Y_r^\eta, X_r^\eta)$. In the second case, x can again be charged to y which contributes 1 to $local\text{-}error^\eta(Y_s^\eta, X_s^\eta)$.

Hence,

$$|E(R, T)| \leq \sum_{j=l-1}^{1} \sum_{i=g^j}^{h^j} local\text{-}error^j(Y_i^j, X_i^j).$$

$$StrEdit(L_a^l \cap Y_a^l, L_a^l \cap Y_a^l) \leq d_g^h + local\text{-}error^l(Y_a^l, X_a^l) + \sum_{j=l-1}^{1}\sum_{i=g^j}^{h^j} local\text{-}error^j(Y_i^j, X_i^j)$$

$$\leq d_g^h + d_g^h\sqrt{24\ln d_g^h} + (l-1)*d_g^h\sqrt{24\ln d_g^h} \leq d_g^h + ld_g^h\sqrt{24\ln d_g^h}.$$

Hence due to phase l, the total edit cost paid is at most

$$\sum_{a=1}^{z^l} StrEdit(L_a^l \cap Y_a^l, L_a^l \cap X_a^l) \leq d + l*d\sqrt{24\ln d}.$$

Now, since we are using a $\beta(n)$-approximation algorithm for StrEdit, we get the total edit cost paid during phase l is at most $\beta(n)(d + l*d\sqrt{24\ln d})$.

For the above bound to be correct, all the $local\text{-}error$ estimates have to be correct. The number of blocks reduces by $\frac{1}{2}$ from one phase to the next. Hence, the total number of local error estimates is $\Theta(z)$. We have considered the iteration such that property (P1) holds for all sequence of blocks (see Lemma 23). Given (P1) holds, since there are a total of $\Theta(z)$ blocks over all the phases, with probability at least $1-\frac{\Theta(z)}{d^3} > 1-\frac{1}{d}$, all the $local\text{-}error$ bounds used in the analysis are correct. Since, (P1) holds with probability at least $(1-\frac{1}{n})$, with probability at least $\left(1-\frac{1}{n}\right)\left(1-\frac{1}{d}\right) > 1-\frac{1}{n}-\frac{1}{d}$, we get a total edit cost paid during phase l is at most $\beta(n)(d + l*d\sqrt{24\ln d})$. ☐

Lemma 25. *The total edit cost paid is at most $O((\log z)^2\beta(n)\sqrt{\ln d})$ with probability at least $1-\frac{1}{n}-\frac{1}{d}$.*

Proof. By Lemma 24, summing up to and including phase l, the total edit cost paid is at most $O(\beta(n)l^2 d\sqrt{\ln d})$ with probability $1-\frac{1}{n}-\frac{1}{d}$ when the number of phases is l. Now each phase reduces the number of blocks at least by a factor of 2. Hence the total number of phases $\leq \lceil \log z \rceil + 1$. Therefore, total edit cost paid is at most $O(\beta(n)(\log z)^2 d\sqrt{\ln d})$. ☐

Theorem 26. *There exists an algorithm that obtains an $O(\beta(n)\sqrt{\ln d}(\log z)^2)$-approximation factor for edit distance computation to* Dyck(s) *for any $s \geq 2$ in $O(n\log n + \alpha(n))$ time with probability at least $\left(1-\frac{1}{n}-\frac{1}{d}\right)$, where there exists an algorithm for* StrEdit *running in $\alpha(n)$ time that achieves an approximation factor of $\beta(n)$.*

Proof. For a particular iteration, each StrEdit is run on a disjoint subsequence. Hence, the running time of the algorithm is $O(n\log_b n + \alpha(n))$. Therefore, from Lemma 25, we get the theorem. ☐

4.2 Improving the Bound to $O(\beta(n)\log z\sqrt{\ln d})$

The above argument can be easily strengthened to improve the approximation factor to $O(\beta(n)\log z\sqrt{\ln d})$.

Lemma 27. *The total edit cost paid is at most $O(\beta(n)\log z\sqrt{\ln d})$ with probability at least $1-\frac{1}{n}-\frac{1}{d}$.*

Proof. Consider any level $l \geq 1$. let $Y_{g^l}^l$ and $X_{g^l}^l$ contains blocks from index $[g^{l-1}, h^{l-1}]$ of level $l-1$, all of which together contain blocks $[g^{l-2}, h^{l-2}]$ from level $l-2$ and so on, finally $[g^1, h^1] = [g, h]$ of blocks from level 1. Let d_g^h denote the optimal edit distance to match $Y_{g^l}^l$ and $X_{g^l}^l$ excluding the symbols that are matched outside of blocks $[g, h]$ by the optimal algorithm.

We want to bound

$$\sum_{j=l}^{1}\sum_{a=g^j}^{h^j} StrEdit(L_a^j \cap Y_a^j, L_a^j \cap X_a^j)$$

Let $R_a^j = L_a^j \cap Y_a^j$ and $T_a^j = L_a^j \cap X_a^j$. Note that either $R_a^j = Y_a^j$ or $T_a^j = X_a^j$.

Let $A(Y_a^j)$ indicate the minimum index of original phase-1 block that Y_a^j contains and $B(X_a^j)$ indicate the maximum index of original phase-1 block that X_a^j contains. Let $D(R_a^j, T_a^j)$ denote all the symbols in R_a^j, T_a^j such that their matching parentheses belong to blocks either outside of index $[A(Y_a^j), B(X_a^j)]$ or they exist at phase j but are not included. Let $E(R_a^j, T_a^j)$ denote all the symbols in R_a^j, T_a^j such that their matching parentheses belong to blocks $[A(Y_a^j), B(X_a^j)]$ but have already been deleted in phases $1, 2, .., j-1$.

Then

$$StrEdit(L_a^j \cap Y_a^j, L_a^j \cap X_a^j) = StrEdit(R_a^j, T_a^j) \le d_{A(Y_a^j)}^{B(X_a^j)} + |D(R_a^j, T_a^j)| + |E(R_a^j, T_a^j)|$$

where, $d_{A(Y_a^j)}^{B(X_a^j)}$ denotes the optimal edit distance to match Y_a^j and X_a^j excluding the symbols that are matched outside of blocks $[A(Y_a^j), B(X_a^j)]$.

Now, by definition if $d_{A(Y_a^j)}^{B(X_a^j)} \ge 2$ then

$$\begin{aligned} |D(R_a^j, T_a^j)| &\le \textit{local-error}^j(Y_a^j, X_a^j) = A_a^{OPT, local^j} \setminus A_a^{local^j} \\ &\le d_{A(Y_a^j)}^{B(X_a^j)} + d_{A(Y_a^j)}^{B(X_a^j)} \sqrt{12 \ln d_{A(Y_a^j)}^{B(X_a^j)}} \le d_{A(Y_a^j)}^{B(X_a^j)} \sqrt{24 \ln d_{A(Y_a^j)}^{B(X_a^j)}}. \end{aligned}$$

Else if $d_{A(Y_a^j)}^{B(X_a^j)} = 1$ then $|D(R_a^j, T_a^j)| \le 2 d_{A(Y_a^j)}^{B(X_a^j)}$.

For each $x \in E(R_a^j, T_a^j)$, consider the largest phase $\eta \in [1, 2, .., j-1]$ such that its matching parenthesis y existed before start of the ηth phase but does not exist after the end of the ηth phase. It is possible, either y belongs to the same block in phase η, say the rth block, or in different blocks, say the rth and the sth block. In the first case, x can be charged to y which contributes 1 to $\textit{local-error}^\eta(Y_r^\eta, X_r^\eta)$. In the second case, x can again be charged to y which contributes 1 to $\textit{local-error}^\eta(Y_s^\eta, X_s^\eta)$.

Therefore, since x cannot belong to multiple R_a^j, T_a^j for a fixed j, we have

$$\sum_{j=l}^{1} \sum_{a=g^j}^{h^j} |E(R_a^j, T_a^j)| \le \sum_{j=l}^{1} \sum_{a=g^j}^{h^j} \textit{local-error}^j(Y_a^j, X_a^j)$$

We also have

$$\sum_{j=l}^{1} \sum_{a=g^j}^{h^j} |D(R_a^j, T_a^j)| \le \sum_{j=l}^{1} \sum_{a=g^j}^{h^j} \textit{local-error}^j(Y_a^j, X_a^j)$$

Therefore,

$$\begin{aligned} \sum_{j=l}^{1} \sum_{a=g^j}^{h^j} StrEdit(L_a^j \cap Y_a^j, L_a^j \cap Y_a^j) &= d_g^h + \sum_{j=l}^{1} \sum_{a=g^j}^{h^j} (|D(R_a^j, T_a^j)| + |E(R_a^j, T_a^j)|) \\ &\le d_g^h + 2 \sum_{j=l}^{1} \sum_{a=g^j}^{h^j} \textit{local-error}^j(Y_a^j, X_a^j) \le d_g^h + 2l d_g^h \sqrt{24 \ln d_g^h} \end{aligned}$$

We assumed $d_g^h \ge 2$, else $\sum_{j=l}^{1} \sum_{a=g^j}^{h^j} StrEdit(L_a^j \cap Y_a^j, L_a^j \cap Y_a^j) \le 5 d_g^h$.

Since, this holds for any level, and $l \le \lceil \log z \rceil + 1$, we get the desired bound stated in the lemma. The probabilistic bound comes from the same argument as in Lemma 24. □

Therefore, we get the improved theorem

Theorem (3)**.** *There exists an algorithm that obtains an $O(\beta(n)\log z\sqrt{\ln d})$-approximation factor for edit distance computation from strings to* $\textsc{Dyck}(s)$ *for any $s \geq 2$ in $O(n\log n + \alpha(n))$ time with probability at least $\left(1-\frac{1}{n}-\frac{1}{d}\right)$, where there exists an algorithm for* $\textsc{StrEdit}$ *running in $\alpha(n)$ time that achieves an approximation factor of $\beta(n)$, and z is the number of blocks.*

4.3 Getting Rid of $\sqrt{\ln d}$-term in the Approximation Factor

We can improve the approximation factor to $O(\beta(n)\log z)$, if we consider $O(n^{\epsilon}\log n)$ iterations instead of $O(\log n)$. We can then use Corollary 14 instead of Lemma 13 to bound the hitting time of the random walk to $\frac{1}{\epsilon}\frac{d^2}{\log d}$, and hence the number of deletions performed by **Random-deletion** process. Local error now improves from $A_a^{OPT,local^i} \setminus A_a^{local^i} = O(d_g^h\sqrt{\ln d_g^h})$ to $A_a^{OPT,local^i} \setminus A_a^{local^i} = O(\frac{1}{\epsilon}d_g^h)$ using the same Chernoff bound argument. Now following the same argument as before and noting that the best known algorithm for $\textsc{StrEdit}$ also runs in $n^{1+\epsilon}$ time returning an $O((\log n)^{\frac{1}{\epsilon}})$ approximation we get the following theorem.

Theorem (4)**.** *For any $\epsilon > 0$, there exists an algorithm that obtains an $O(\frac{1}{\epsilon}\log z(\log n)^{\frac{1}{\epsilon}})$-approximation factor for edit distance computation to* $\textsc{Dyck}(s)$ *for any $s \geq 2$ in $O(n^{1+\epsilon})$ time with high probability, and z is the number of blocks.*

Note. Due to local computations, it is possible to parallelize this algorithm.

5 Memory Checking Languages

Our algorithm and analysis for $\textsc{Dyck}(s)$ gives a general framework which can be applied to the edit distance computation problem to many languages. Here we illustrate this by considering memory checking languages such as PQ, $\textsc{Queue}$, $\textsc{Stack}$ and $\textsc{Deque}$. These languages check whether a given transcript in memory corresponds to a particular data structure such as priority queue, queue, stack and double-ended queue respectively. Formally, we ask the following question, we observe a sequence of n updates and queries to (an implementation of) a data structure, and we want to report whether or not the implementation operated correctly on the instance observed and if not what is the minimum number of changes need to be done to make it a correct implementation. A concrete example is to observe a transcript of operations on a priority queue: we see a sequence of insertions intermixed with items claimed to be the results of extractions, and the problem is to detect a way to minimally change the transcript to make it correct. This is the model where checker is invasive and allowed to make changes. Most of the prior literature considered such invasive checkers [2, 11, 13, 29].

Stack. Let $\textsc{Stack}(s)$ denote the language over interaction sequences of s different symbols that correspond to stack operations. Let $ins(u)$ correspond to an insertion of u to a stack, and $ext(u)$ is an extraction of u from the stack. Then $\sigma \in Stack$ iff σ corresponds to a valid transcript of operations on a stack which starts and ends empty. That is, the state of the stack at any step j can be represented by a string S^j so that $S^0 = \phi$, $S^j = uS^{j-1}$ if $\sigma_j = ins(u)$, $uS^j = S^{j-1}$ if $\sigma(j) = ext(u)$ and $S^n = \phi$.

It is easy to see that $\textsc{Stack}(s)$=$\textsc{Dyck}(s)$ by assigning u to $ins(u)$ and $\bar{u}$ to $ext(u)$. Hence, we can employ the algorithm for $\textsc{Dyck}(s)$ to estimate the edit distance to $\textsc{Stack}(s)$ efficiently.

Priority Queue. Let PQ(s) denote the language over interaction sequences of s different symbols that correspond to priority queue operation. That is the state of priority queue at any time j can be represented by a multiset M^j such that $M^0 = M^n = \emptyset$. $M^j = M^{j-1}u$ if $\sigma_j = ins(u)$ and $M^j = M^{j-1} \setminus \{\min(M^{j-1})\}$ if $\sigma_j = ext(\nu)$. We view $ins(u)$ as u and $ext(u) = \bar{u}$, but each u now has a priority. Note that σ can be represented as $Y_1X_1Y_2X_2....Y_zX_z$ where each $Y_i \in T^+$ and each $X_i \in \bar{T}^+$.

We proceed with *Random-deletion* but when the process starts considering symbols from X_k, we sort the prefix of open parenthesis by priority, so that highest priority element is at the stack top. After that using the boundaries computed by *Random-deletion*, one can employ the main refined algorithm from Section 4. It can be verified by employing the same analysis that this results in a polylog-approximation algorithm for PQ in $\tilde{O}(n)$ time.

Queue. Let QUEUE(s) denote the language over interaction sequences of s different symbols that correspond to queue operations. That is the state of queue at any time j can be represented by a string Q^j such that $Q^0 = Q^n = \phi$. $Q^j = Q^{j-1}u$ if $\sigma_j = ins(u)$ and $uQ^j = Q^{j-1}$ if $\sigma_j = ext(u)$. Now QUEUE(s) is nothing but a PQ(s) with priority given by time of insertion, earlier a symbol is inserted, higher is its priority. Therefore using the algorithm for PQ(s), we can estimate the edit distance to QUEUE(s) efficiently.

Deque. Let DEQUE(s) denote the language over interaction sequences that correspond to double-ended queues. That is, there are now two types of insert and extract operations, one operation for the head and one for the tail. We now create two strings σ_1 and σ_2 where σ_1 contains all the insertions and only extractions from the tail, whereas σ_2 contains again all the insertions and only extractions from the head. σ_1 is created according to STACK protocol, whereas σ_2 is created according to QUEUE protocol. We start running *Random-deletion* on both σ_1 and σ_2 simultaneously as follows. If *Random-deletion* is comparing (may lead to either matching or deletion) an extraction σ_j in σ_1 and $\sigma_{j'}$ in σ_2, and $j < j'$, then we take one step of *Random-deletion* in σ_1 and if $j' < j$ then we take one step of *Random-deletion* in σ_2. If *Random-deletion* deletes an insertion from σ_1 which still exists in σ_2, we delete it from σ_2 as well and vice-versa. Once, we can perform *Random-deletion*, we can employ our main algorithm and the same analysis to show that this gives an $O(poly \log n)$ approximation.

6 Conclusion & Future Directions

In this paper, we give the first nontrivial approximation algorithm for edit distance computation to DYCK(s) that runs in near-linear time. DYCK(s) is a fundamental context free language, and a restricted subset of this language is known as the hardest in the context-free class. The techniques that we develop for DYCK(s) can be employed for edit distance computation to other larger class of languages, which we illustrate by considering languages accepted by common data structures. There are several open questions that arise from this work.

1. Is it possible to characterize the general class of grammars for which **Random-deletion** and/or its subsequent refinements can be applied?

2. The lower bound result of Lee [24] precludes an exact algorithm (as well as one with nontrivial multiplicative approximation factor) for language edit distance problem for general context free grammars in time less than boolean matrix multiplication. Does this lower bound also hold for DYCK(s)? Is boolean matrix multiplication time enough to compute language edit distance for any arbitrary context free grammar? Is it possible to achieve nontrivial approximation guarantees when the time required for parsing a grammar is allowed?

3. Currently there is a gap of $\log z$ in the approximation factors of string and DYCK(s) edit distance problems. Is it possible to get rid off this gap, or establish the necessity of it? What happens in other computation models such as streaming?

4. Finally, our multi-phase random walk technique can be very useful to provide a systematic way to

speed up dynamic programming algorithms for sequence alignment type problems. It will be interesting to understand whether this method can lead to a better bound for string edit distance computation.

7 Acknowledgment

The author would like to thank Divesh Srivastava and Flip Korn for asking this nice question which led to their joint work [20], Arya Mazumdar for many helpful discussions and Mikkel Thorup for comments at an early stage of this work.

References

[1] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4), 1972.

[2] Miklós Ajtai. The invasiveness of off-line memory checking. STOC '02, 2002.

[3] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM J. Comput.*, 30(6), December 2001.

[4] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *FOCS*. IEEE, 2010.

[5] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. STOC '09, pages 199–204, 2009.

[6] Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. Approximating edit distance efficiently. FOCS '04, 2004.

[7] Tuğkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. SODA '06, pages 792–801, 2006.

[8] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. FOCS '91, 1991.

[9] Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. FOCS '10, 2010.

[10] Noam Chomsky and Marcel Paul Schützenberger. *The Algebraic Theory of Context-Free Languages*. Studies in Logic. North-Holland Publishing, Amsterdam, 1963.

[11] Matthew Chu, Sampath Kannan, and Andrew McGregor. Checking and spot-checking the correctness of priority queues. ICALP'07, 2007.

[12] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM J. Comput.*, 31(6), June 2002.

[13] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? TCC '09, 2009.

[14] C. N. Fischer and J. Mauney. On the role of error productions in syntactic error correction. *Comput. Lang.*, 5(3-4), January 1980.

[15] Charles N. Fischer and Jon Mauney. A simple, fast, and effective ll(1) error repair algorithm. *Acta Inf.*, 29(2), April 1992.

[16] Steven Grijzenhout and Maarten Marx. The quality of the xml web. *Web Semant.*, 19, March 2013.

[17] R.R Gutell, J.J. Cannone, Z Shang, Y Du, and M.J Serra. A story: unpaired adenosine bases in ribosomal rnas. J Mol Biol, 2010.

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

[19] Ik-Soon Kim and Kwang-Moo Choe. Error repair with validation in lr-based parsing. *ACM Trans. Program. Lang. Syst.*, 23(4), July 2001.

[20] Flip Korn, Barna Saha, Divesh Srivastava, and Shanshan Ying. On repairing structural problems in semi-structured data. VLDB'13, 2013.

[21] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[22] Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. MFCS'11, 2011.

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

[24] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1), January 2002.

[25] F Florence MacWilliams and Neil Sloane. *The Theory of Error-correcting Codes: Part 2*, volume 16. Elsevier, 1977.

[26] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. STOC '10, 2010.

[27] Bruce McKenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17, 1995.

[28] Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54, 1995.

[29] Moni Naor and Guy N. Rothblum. The complexity of online memory checking. *J. ACM*, 56(1), February 2009.

[30] Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. *J. ACM*, 54(5), October 2007.

[31] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Struct. Algorithms*, 22(1), January 2003.

[32] S. Rajasekaran, S. Al Seesi, and R. Ammar. Improved algorithms for parsing esltags:a grammatical model suitable for rna pseudoknots. Tran. Comp. Bio. and Bioinformatics, 2010.

[33] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. FOCS '96, 1996.

[34] Remco van der Hofstad and Michael Keane. An elementary proof of the hitting time theorem. *The American Mathematical Monthly*, 115(8), 2008.

[35] Balaji Venkatachalam, Dan Gusfield, and Yelena Frid. Faster algorithms for rna-folding using the four-russians method. In *WABI*, 2013.

8 Appendix

All missing proofs are provided here.

Lemma (6)**.** *For any string $\sigma \in (T \cup \bar{T})^*$, $OPT(\sigma) \leq OPT_d(\sigma) \leq 2OPT(\sigma)$.*

Proof. Clearly, $OPT(\sigma) \leq OPT_d(\sigma)$, since to compute $OPT(\sigma)$, all edit operations: insertion, deletion and substitution were allowed but for $OPT_d(\sigma)$ only deletion was allowed and hence the number of edit operations can only increase.

To prove the other side of inequality, consider each type of edits that are done on σ to compute $OPT(\sigma)$. First consider only the insertions. Let the positions of insertions are immediately after the indices $i_1, i_2, ...i_l$. These insertions must have been done to match symbols, say at positions $j_1, j_2, ..., j_l$, otherwise, it is easy to refute that $OPT(\sigma)$ is not optimal. Instead of the l insertions, we could have deleted the symbols at $j_1, j_2, ..., j_l$ with equal cost. Therefore, all insertions can be replaced by deletions at suitable positions without increasing $OPT(\sigma)$.

Next, consider the positions where substitutions have happened. Let these be $i'_1, i'_2, ..., i'_{l'}$, for some $l' \geq 0$. If $l' = 0$, then $OPT(\sigma) = OPT_d(\sigma)$. Otherwise, let $l' \geq 1$. Consider the position i'_1. After substitution at position i'_1, the new symbol at i'_1 must match a symbol at some position j, such that either $j \in \{i'_2, i'_3, ...i'_{l'}\}$ or j is outside of this set. When $j \in \{i'_2, i'_3, ...i'_{l'}\}$, instead of two substitutions at i'_1 and j, one can simply delete the symbols at these two positions maintaining the same edit cost. When j does not belong to $\{i'_2, i'_3, ...i'_{l'}\}$, instead of one substitution at position i'_1, one can do two deletions at positions i'_1 and j. Therefore each substitution operation can be replaced by at most two deletions. Whereas each insertion can be replaced by a single deletion. Continuing in this fashion, overall, this ensures a 2 approximation, that is $OPT_d(\sigma) \leq 2OPT(\sigma)$. □

Lemma (7)**.** *There exists an optimal algorithm that makes a single scan over the input pushing open parentheses to stack and on observing a close parenthesis, the algorithm compares it with the stack top. If the symbols match, then both are removed from further consideration, otherwise one of the two symbols is deleted.*

Proof. The statement is true when there is 0 error. Suppose the statement is true when the number of minimum edits required is d. Now consider any string for which the minimum edit distance to DYCK language is $d + 1$. A stack based algorithm must find a mismatch at least once between stack top and the current symbol in the string. Consider the first position where it happens. Suppose, without loss of generality, at that point, the stack top contains "$($" and the current symbol is "$]$". Any optimal algorithm that does minimum number of edits must change at least one of these symbols. There are only two alternatives, (a) delete "$($", or (b) delete "$]$". Our stack based algorithm can take exactly the same option as the optimal algorithm. This reduces the number of edits required in the remaining string to make it well-balanced and the induction hypothesis applies. □

Lemma (9)**.** *If at time t (up to and including time t), the number of indices in A_t^{OPT} that the optimal algorithm deletes is d_t and the number of correct and wrong moves are respectively c_t and w_t then $|A_t^{OPT} \setminus A_t| \leq d_t + w_t - c_t$.*

Proof. Clearly, the lemma holds at time $t = 0$, since $d_0 = c_0 = w_0 = 0$ and $A_0 = A_0^{OPT} = \emptyset$.

Suppose, the lemma holds up to and including time $t-1$. We now consider time t. Let at time t, the algorithm compares an open parenthesis $\sigma[i]$ with a close parenthesis $\sigma[j]$, $i < j$. The following cases need to be considered.

Case 1. $\sigma[i]$ is matched with $\sigma[j]$. $c_t = c_{t-1}, w_t = w_{t-1}$.

Subcase 1. The optimal algorithm also matches $\sigma[i]$ with $\sigma[j]$, hence $d_t = d_{t-1}$. Now $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j\}$ and $A_t = A_{t-1} \uplus \{i, j\}$. So we have $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $= d_t + w_t - c_t$.

Subcase 2. The optimal algorithm does not match $\sigma[i]$ with $\sigma[j]$. It is not possible that the optimal algorithm matches $\sigma[i]$ with $\sigma[j']$, $j' > j$ and also matches $\sigma[j]$ with $\sigma[i']$, $i' < j$ simultaneously due to the property of well-formedness.

- First consider that $\sigma[i]$ and $\sigma[j]$ are both matched with some symbols in the optimal algorithm. So $d_t = d_{t-1}$. Let $\sigma[i]$ be matched with $\sigma[j']$ and $\sigma[j]$ be matched with $\sigma[i']$. Then either $(a) i < j' < i' < j$, or $(b) i < i' < j < j'$, or $(c) i' < i < j' < j$.
  - For (a) $i, j \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT}$. Also, $A_t = A_{t-1} + \{i, j\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 2 \leq d_{t-1} + w_{t-1} - c_{t-1} - 2 = d_t + w_t - c_t - 2$.
  - For (b), $j \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j'\}$. Also, $A_t = A_{t-1} \uplus \{i, j\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}|$ (decreases 1 due to j and increases 1 due to j') and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $= d_t + w_t - c_t$.
  - For (c), $i \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i', j\}$. Also, $A_t = A_{t-1} \uplus \{i, j\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}|$ (decreases 1 due to i and increases 1 due to i') and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $= d_t + w_t - c_t$.
- Now consider that one of $\sigma[i]$ or $\sigma[j]$ gets deleted. Assume, w.l.o.g, that $\sigma[i]$ is deleted (exactly similar analysis when only $\sigma[j]$ is deleted). So $d_t = d_{t-1} + 1$. Then if $\sigma[j]$ is matched with $\sigma[i']$ either $(a') i < i' < j$ or $(b') i' < i < j$.
  - For (a'), $j \in A_{t-1}^{OPT}$, $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i\}$. Also, $A_t = A_{t-1} \uplus \{i, j\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1$ (decreases 1 due to j) and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1 \leq d_{t-1} + w_{t-1} - c_{t-1} - 1$(by induction hypothesis) $< d_t + w_t - c_t$.
  - For (b'), $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j, i'\}$. Also, $A_t = A_{t-1} \uplus \{i, j\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1$ (increases 1 due to i') and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1 \leq d_{t-1} + w_{t-1} - c_{t-1} + 1$(by induction hypothesis) $= d_t + w_t - c_t$.
- Now consider that both $\sigma[i]$ and $\sigma[j]$ are deleted. $d_t = d_{t-1} + 2$. $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j\}$. Also, $A_t = A_{t-1} \uplus \{i, j\}$. Hence, $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1} < d_t + w_t - c_t$.

Case 2. $\sigma[i]$ is not matched with $\sigma[j]$ and $\sigma[i]$ is deleted.

- First consider that in the optimal algorithm, $\sigma[i]$ and $\sigma[j]$ are both matched with some symbols. So $d_t = d_{t-1}$. Let $\sigma[i]$ be matched with $\sigma[j']$ and $\sigma[j]$ be matched with $\sigma[i']$. Then either $(a) i < j' < i' < j$, or $(b) i < i' < j < j'$, or $(c) i' < i < j' < j$.
  - For (a) $i, j \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1 \leq d_{t-1} + w_{t-1} - c_{t-1} - 1 = d_t + w_t - c_t$, since $c_t = c_{t-1} + 1$.
  - For (b), $j \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j'\}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1$ and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1 \leq d_{t-1} + w_{t-1} - c_{t-1} + 1$(by induction hypothesis) $= d_t + w_t - c_t$, since $w_t = w_{t-1} + 1$.

- For (c), $i \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1$ (decreases 1 due to i) and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1 \leq d_{t-1} + w_{t-1} - c_{t-1} - 1$(by induction hypothesis) $= d_t + w_t - c_t$, since $c_t = c_{t-1} + 1$.

- Now consider that one of $\sigma[i]$ or $\sigma[j]$ gets deleted by the optimal algorithm. Assume, first, that $\sigma[i]$ is deleted. So $d_t = d_{t-1} + 1$. Then if $\sigma[j]$ is matched with $\sigma[i']$ either $(a')i < i' < j$ or $(b')i' < i < j$.

  - For (a'), $j \in A_{t-1}^{OPT}$, $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i\}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}|$ and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $< d_t + w_t - c_t$, since $d_t = d_{t-1} + 1$, $c_t = c_{t-1} + 1$ and $w_t = w_{t-1}$.
  - For (b'), $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i\}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}|$ and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $< d_t + w_t - c_t$, since $d_t = d_{t-1} + 1$, $c_t = c_{t-1} + 1$ and $w_t = w_{t-1}$.

  Assume now that the optimal algorithm only deletes $\sigma[j]$. So, $d_t = d_{t-1} + 1$, because *Random-deletion* takes action on $\sigma[i]$ and the optimal algorithm matches it. If $\sigma[i]$ is matched with $\sigma[j']$ either $(a'')i < j' < j$, or $(b'')i < j < j'$.

  - For (a''), $i \in A_{t-1}^{OPT}$, $A_t^{OPT} = A_{t-1}^{OPT}$ and $A_t = A_{t-1} \uplus \{i\}$.Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1$ and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1 \leq d_{t-1} + w_{t-1} - c_{t-1} - 1$(by induction hypothesis) $< d_t + w_t - c_t$, since $c_t = c_{t-1} + 1$ and $w_t = w_{t-1}$.
  - For (b''), $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i, j'\}$ and $A_t = A_{t-1} \uplus \{i\}$.Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1$ and thus $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| + 1 \leq d_{t-1} + w_{t-1} - c_{t-1} + 1$(by induction hypothesis) $< d_t + w_t - c_t$, since $w_t = w_{t-1} + 1$ and $c_t = c_{t-1}$.

- Now consider that both $\sigma[i]$ and $\sigma[j]$ get deleted by the optimal algorithm. $d_t = d_{t-1} + 1$, since *Random-deletion* takes only action on $\sigma[i]$. $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{i\}$. Also, $A_t = A_{t-1} \uplus \{i\}$. Hence, $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1} = d_t + w_t - c_t$, since $d_t = d_{t-1} + 1$ and $c_t = c_{t-1} + 1$.

Case 3. $\sigma[i]$ is not matched with $\sigma[j]$ and $\sigma[j]$ is deleted.
Same as Case 2.

Case 4. Now consider the case that at time t, the stack is empty and the current symbol in the string is $\sigma[j]$. In that case *Random-deletion* deletes $\sigma[j]$ and the move is correct. So $c_t = c_{t-1} + 1$ and $w_t = w_{t-1}$. We have $A_t = A_{t-1} \uplus \{j\}$. If the optimal algorithm also deletes $\sigma[j]$ then $A_t^{OPT} = A_{t-1}^{OPT} \uplus \{j\}$ and $d_t = d_{t+1} + 1$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| \leq d_{t-1} + w_{t-1} - c_{t-1}$(by induction hypothesis) $= d_t + w_t - c_t$, since $d_t = d_{t-1} + 1, c_t = c_{t-1} + 1$. On the other hand, if the optimal algorithm matches $\sigma[j]$ with some $\sigma[i']$, $i' < j$, then $j \in A_{t-1}^{OPT}$ and $A_t^{OPT} = A_{t-1}^{OPT}$. Hence $|A_t^{OPT} \setminus A_t| = |A_{t-1}^{OPT} \setminus A_{t-1}| - 1 \leq d_{t-1} + w_{t-1} - c_{t-1} - 1$(by induction hypothesis) $= d_t + w_t - c_t - 2$, since $d_t = d_{t-1}, c_t = c_{t-1} + 1$.

Case 5. Now consider the case that at time t, the input is exhausted and the algorithm considers $\sigma[i]$ from the stack top. In that case *Random-deletion* deletes $\sigma[i]$. Then, again by similar analysis as in the previous case, the claim is established. □

8.1 Pseudocode of Refined Algorithm

Input $\sigma = Y_1 X_1 Y_2 X_2 .... Y_z X_z$
Initialization: $\sigma' = \sigma$; $i = 1$
while $a \leq z$ **do**
  $N_{a,min} = \infty$; $Z_{a,min} = \emptyset$, $startIn = i$
  for $count = 1; count < 2\log n; count++;$ **do**
    $Z = \emptyset$, $d_a = 0$, $i = startI$
    while processing on X_a is not completed **do**
      if $\sigma'[i] == o$ **then**
        Insert $\sigma'[i]$ in stack
        $i++$
      else if $\sigma'[i]$ matches top of stack **then**
        match $\sigma'[i]$ with top of stack, append top of stack to Z, and remove both of them
        $i++$
      else
        with probability $\frac{1}{2}$ each select one of $\sigma'[i]$ or top of stack to be deleted
        if top of stack is selected **then**
          append that to Z
        end if
        delete the selected symbol
        $d_a = d_a + 1$
        if $\sigma'[i]$ is deleted **then**
          $i++$
        end if
      end if
    end while
    if $d_a \leq N_{a,min}$ **then**
      $N_{i,min} = d_a$; $Z_{a,min} = Z$, $endIn = i - 1$
    end if
    Start again with σ'
  end for
  Remove $(Z_{a,min}, X_a)$ from σ'
  $(R_a, T_a) = StrEdit(Z_{a,min}, X_a)$
  Replace $(Z_{a,min}, X_{a,min})$ in σ by (R_a, T_a)
  $a = a + 1$, $i = endIn + 1$
end while
if there are excess open parenthesis in σ' **then**
  Delete those corresponding open parenthesis from σ;
end if
return σ

Figure 2: Improved Edit Distance Computation to DYCK(s)