Improved bounds for testing Dyck languages

Eldar Fischer[1], Frédéric Magniez[2], and Tatiana Starikovskaya[3]

[1]Technion - Israel Institute of Technology, Israel, eldar@cs.technion.ac.il
[2]CNRS, IRIF, Univ Paris Diderot, France, magniez@irif.fr
[3]IRIF, Univ Paris Diderot, France, tat.starikovskaya@gmail.com

Abstract

In this paper we consider the problem of deciding membership in Dyck languages, a fundamental family of context-free languages, comprised of well-balanced strings of parentheses. In this problem we are given a string of length n in the alphabet of parentheses of m types and must decide if it is well-balanced. We consider this problem in the property testing setting, where one would like to make the decision while querying as few characters of the input as possible.

Property testing of strings for Dyck language membership for $m = 1$, with a number of queries independent of the input size n, was provided in [Alon, Krivelevich, Newman and Szegedy, SICOMP 2001]. Property testing of strings for Dyck language membership for $m \geq 2$ was first investigated in [Parnas, Ron and Rubinfeld, RSA 2003]. They showed an upper bound and a lower bound for distinguishing strings belonging to the language from strings that are far (in terms of the Hamming distance) from the language, which are respectively (up to polylogarithmic factors) the $2/3$ power and the $1/11$ power of the input size n.

Here we improve the power of n in both bounds. For the upper bound, we introduce a recursion technique, that together with a refinement of the methods in the original work provides a test for any power of n larger than $2/5$. For the lower bound, we introduce a new problem called Truestring Equivalence, which is easily reducible to the 2-type Dyck language property testing problem. For this new problem, we show a lower bound of n to the power of $1/5$.

1 Introduction

1.1 Background

Initially identified as one of the ingredients for the proof of the PCP theorem [3], property testing is nowadays one of the successful paradigms of computation for handling massive data sets. In property testing one would like to decide whether an input has a global property by performing only few local checks. The goal is to distinguish with sufficient confidence the inputs which satisfy the property from those that are far from satisfying it. In this sense, property testing is a notion of approximation for the corresponding decision problem. Property testers, under the name of self-testers and with a slightly different objective, were first considered for programs computing functions with some algebraic properties [6, 7, 22, 21]. The notion in its full generality was defined by Goldreich, Goldwasser and Ron, and successfully applied to topics including testing properties of graphs [16, 17], monotonicity [15], group and field operations [12], geometrical objects [10], formal languages [1], and probability distributions [5]. The setting of property testing has also been addressed for quantum computers, see [19] for a survey.

Formally, property testers have random access to their input in the query model. They can read the input, one piece at a time, by submitting a query with the selected index. Ideally, property testers should perform a number of queries that depends only on the approximation parameter (and not on the input length n), but also an algorithm making a number of queries that is sublinear in n (for every fixed approximation parameter) is considered a legitimate property test. Whereas the complexity of formal language membership is quite well-understood, both in term of space and time complexity (see for instance [25]), even on parallel architectures [11] and in the streaming model [18], very little is known in the context of property testing, except when both models of streaming and property testing are combined [9, 14].

The study of property testers for formal languages was initiated by Alon et al. [1] under the Hamming distance. In this context, two strings of length n are ε-far if the Hamming distance between them is larger than εn. An ε-tester for a language L must distinguish, with probability at least $2/3$, the strings that are in L from those that are ε-far from L, using as few queries as possible. Alon et al. showed that regular languages, as well as the Dyck language D_1 of well-parenthesized expressions with a single type of parentheses, can be ε-tested with a number of queries independent of the input size n. However, by the work of [20] the query complexity of an ε-tester for Dyck languages D_m with $m \geq 2$ types of parentheses is between $\Omega(n^{1/11})$ and $\tilde{O}(n^{2/3})$, and by [1] it becomes linear, even for one type of parentheses, when one-sided error is required. When the distance allows sufficient modifications of the input, such as moves of arbitrarily large factors, it has been shown that any context-free language is testable with a constant number of queries [13].

1.2 Motivation and our results

Dyck languages are not only some of the simplest context free languages, but they are also universal in some sense, since any context-free language can be expressed as an intersection of a regular language with D_m, for some integer m and up to some homomorphic map (Chomsky-Schützenberger representation theorem [4]). Moreover, Dyck languages have been used in many real life applications over the years, and some of their extensions, such as visibly pushdown languages or nested strings [2], are heavily used to handle semi-structured documents such as massive databases, or to capture safety properties of programs from their execution traces.

Motivated by the new applications and the prevalence of massive data, there is a renewed interest in studying the complexity of testing membership in context free languages and in particular in

Dyck languages. As an illustration, we mention some of the recent works pioneered by Saha for estimating the edit distance of a string to a Dyck language [23] and to context free languages [24, 8].

In this paper we revisit the complexity of property testing for Dyck languages D_m, with $m \geq 2$, and improve the previously known upper and lower bounds (respectively $\tilde{O}(n^{2/3})$ and $\Omega(n^{1/11})$), significantly narrowing the gap between the two. Our contribution is twofold:

1. Our first main result consists of a new property testing algorithm for Dyck languages. In particular, we show that for any $m \geq 2$ and for any constant $\delta, \varepsilon > 0$ there is an ε-tester with complexity $\tilde{O}(n^{2/5+\delta})$, up to some polylogarithmic factors (Corollary 3.4).
2. Our second main result is an improved lower bound of $\Omega(n^{1/5})$ (Corollary 5.2) for some constant ε. To show this, we introduce a new formalism which can potentially simplify further establishments of lower bounds in property testing.

1.3 Overview of the paper

New algorithms. In order to improve the previously known algorithms, we introduce two recursion techniques, that together with a refinement of the methods from previous works provide a better complexity. We start with an easy reduction from D_m-MEMBERSHIP testing to D_m-CONSISTENCY testing (Section 2, Lemma 2.4), where the later problem simply asks if the input string is (close to) a substring of a member of D_m. The reduction is built upon the tester for D_1-MEMBERSHIP of [1], and was implicit in the analysis of the tester of [20]. Next, in Section 3, we show an algorithm for D_m-CONSISTENCY testing. The algorithm (Algorithm 2) partitions the input string into non-overlapping blocks. Note that if the input string is D_m-consistent, each of the blocks is D_m-consistent as well. A natural part of the algorithm therefore consists of selecting several blocks at random and testing if they are D_m-consistent. Instead of querying all characters of the selected blocks as in [20], we design a careful analysis allowing a recursive call here (Theorem 3.3).

Inter-block matching. We call the parentheses of a block that must be matched with parentheses in a different block *excess*. The crucial part of the algorithm consists of checking that the excess parentheses of the blocks can be matched correctly. In Section 2.3 we construct a dependency graph in order to identify candidate pairs of blocks having many matching excess parentheses between them. This graph is an approximation of the *matching graph* introduced in [20] and it can be computed efficiently (Algorithm 1). We provide a new property of this graph in the context of D_m-consistency testing. Namely, the total weight of its edges accounts for almost all excess parentheses of the blocks that are not excess in the input string (Lemma 2.12). Once such a candidate pair of blocks has been identified, we (approximately) locate their substrings containing the excess parentheses that are required to match. We now need to check whether the excess parentheses of one substring can be matched with the excess parentheses of the second substring. This task is performed by an external algorithm (Algorithm 3), which is itself recursive. We call this task SUBSTRING ε-MATCHING.

Substring ε-matching. Our solution for SUBSTRING ε-MATCHING exploits a new methodology that combines a recursive approach and the birthday paradox. In order to recurse, we would like to divide each substring into smaller subblocks, and recurse on a random pair of the subblocks that contain matching parentheses. However, since we could only locate the substrings approximately, it is hard to identify efficiently the pairs of subblocks to be tested. We overcome this technical hurdle by trying to guess the real borders and testing each of our guesses. Potentially, this can be very expensive in terms of query complexity. To avoid this, our solution uses the birthday paradox. The idea is not to query a separate subset of subblocks for each pair of tested substrings, but to query a square root number of subblocks from both substrings and then re-use them to test each

of our guesses. In the end, we show that Algorithm 3 solves SUBSTRING ε-MATCHING on n-length substrings with query complexity $O(n^{\frac{1}{2}+\delta})$, for any constant $0 < \delta \leq 1/2$ (Theorem 3.2).

New lower bound and methodology. For the improved lower bound (Corollary 5.2), we first introduce a new problem called TRUESTRING EQUIVALENCE (see Section 2), that highlights one particular aspect of testing Dyck languages. In this problem we must decide if two given binary strings with dummy characters "$\diamond$" are equal after deleting all "$\diamond$" characters. The "$\diamond$" characters hide the indexes of the meaningful bits, just as the indexes of excess parentheses are hidden in a parenthesized expression. After a quick reduction from TRUESTRING EQUIVALENCE to D_m-MEMBERSHIP (Lemma 2.5), we proceed to prove a bound for TRUESTRING EQUIVALENCE using the traditional Yao's method (Theorem 5.1) in Section 5. Namely, we produce a distribution $\mathcal{D}_P$ over inputs satisfying TRUESTRING EQUIVALENCE, and a distribution $\mathcal{D}_N$ over inputs that are (mostly) far from satisfying TRUESTRING EQUIVALENCE (Lemma 5.4), and then show that any (possibly adaptive) deterministic algorithm will have only a small difference in its acceptance probability when it is fed either an input drawn according to $\mathcal{D}_P$ or one drawn according to $\mathcal{D}_N$ (Lemma 5.5).

A new formalism. To facilitate the analysis of the behaviour of a deterministic algorithm $\mathcal{A}$, when fed an input drawn according to either distribution, we formalize the technique (at times used informally) of "revealing unrequested information to the algorithm" for the purpose of separating the "irrelevant" probabilistic dependencies, and allowing a probabilistic analysis that exploits the remaining "relevant independency" between the participating random variables.

For this purpose we define the notion of a *super-oracle*, which in response to a requested query may output additional information about the input in addition to the query itself (essentially adding additional queries to the requested query and answering all of them). Then, we analyze the distribution over the transcript of any algorithm that is run against the super-oracle, when the input is drawn by either $\mathcal{D}_P$ or $\mathcal{D}_N$.

A special event that we define below is an event in which the super-oracle provides the input in its entirety, essentially "giving-up" on confounding the algorithm. For a carefully chosen trigger for this event, we show that the distributions of the algorithm's behaviour (with respect to either $\mathcal{D}_P$ or $\mathcal{D}_N$) both "underlie" a common distribution (by which we mean that all outcome probabilities outside the giving-up event are bounded by the respective probabilities of the common distribution), while the probability of the giving-up event itself is small. This implies that the difference between the respective acceptance probabilities is also small, which allows us to invoke Yao's method to conclude the lower bound argument.

2 Preliminaries

2.1 Basic definitions and some reductions

Hereafter n will denote the input size, and $\tilde{O}(f(n))$ stands for $O(f(n) \cdot \operatorname{polylog} n)$. The *distance* between two strings T, T' of length n is the Hamming distance, that is the number of indexes in which they differ. We say that T, T' are *ε-close* when their distance is at most εn, and that they are *ε-far* otherwise.

Definition 2.1 (ε-Tester)**.** *Let L be a language over a constant-size alphabet. A randomized algorithm A is an* ε-tester for L *with bounded error $\eta \geq 0$, if A accepts all inputs $T \in L$ with probability at least $1 - \eta$, and rejects all inputs T that are ε-far from all members of L with probability at least $1 - \eta$.*

Usually, the notion of property testing is studied in the context of *query complexity*. In this model, the algorithm is given the size of the input, but not the input string itself: the algorithm can only access the string by querying it locally, one character at a time. The query complexity of the algorithm is defined as the number of performed queries. In this work we study the worst case query complexity of the algorithms and disregard other notions of space and time complexity.

The Dyck language D_m is the language of strings of properly balanced parentheses of m types. For example, a string "$(_0(_1)_1)_0$" is in D_2, while "$(_0(_1)_0)_0$" and "$(_0(_1)_1(_0$" are not. A string S is *D_m-consistent*, if it is a substring of a string $S' \in D_m$. We now define the three main problems that we will consider, and state some reductions between them. The notion of ε-testing is implicitly extended to those problems by considering the respective languages they define.

D_m-MEMBERSHIP(n)
Input: String of even length n on an alphabet of parentheses of m types
Output: Decide if it is in D_m

D_m-CONSISTENCY(n)
Input: String of length n on an alphabet of parentheses of m types
Output: Decide if it is D_m-consistent

The last problem is defined in a slightly different but related context. Given a string $w \in \{0, 1, \diamond\}^*$, its *truestring* $T(w)$ is the subsequence resulting from deleting all $\diamond$ characters. Two strings w and v are called *truestring equivalent* if $T(w) = T(v)$.

TRUESTRING EQUIVALENCE(n)
Input: Two strings of length n over alphabet $\{0, 1, \diamond\}$
Output: Decide if they are truestring equivalent

We will need a tester for D_1-MEMBERSHIP by Alon et al. [1].

Lemma 2.2 ([1])**.** *There is an ε-tester for D_1-MEMBERSHIP(n) with bounded error $1/6$ and query complexity $O(\varepsilon^{-2}\log(1/\varepsilon))$.*

Definition 2.3 ([20])**.** *For a string S on the alphabet of parentheses of m types, let $\mu(S)$ be a string obtained from S by removing the types of the parentheses.*

Let k and ℓ be the smallest integers such that $({}^k\mu(S))^\ell \in D_1$. We call $e_1(S) = k$ the excess number of closing parentheses *in S, and $e_0(S) = \ell$ the* excess number of opening parentheses *in S.*

If S is a substring of the input string T, its excess numbers indicate how many parentheses cannot be matched with other parentheses in S and must be matched with parentheses outside S. Such parentheses are called excess parentheses *of S.*

For example, if $S =$ "$)_0(_1)_1)_0(_1)_1$", then $\mu(S)$ is "$)())()$", $e_1(S) = 2$ and $e_0(S) = 0$, and the excess parentheses are the first and the fourth ones. Let $n_0(S')$ be the number of opening parentheses in a substring S' of S, and $n_1(S'')$ be the number of closing parentheses in a substring S'' of S. It is not hard to see that the following equations hold, where we assume that the empty prefix and the empty suffix are included:

$$e_1(S) = \max_{S'-\text{prefix of } S} \big(n_1(S') - n_0(S')\big), \quad \textit{and} \quad e_0(S) = \max_{S''-\text{suffix of } S} \big(n_0(S'') - n_1(S'')\big) \tag{1}$$

We can now state our reductions.

Lemma 2.4. *Given an ε-tester A for D_m-CONSISTENCY(n) with bounded error $1/6$, one can design an $\Theta(\varepsilon)$-tester B for D_m-MEMBERSHIP(n) with bounded error $1/3$ and query complexity equal to that of A with an additional term of $O(\varepsilon^{-2}\log(1/\varepsilon))$.*

Proof. Our tester for D_m-MEMBERSHIP(n) on T consists of two steps: first, we apply the tester for D_1-MEMBERSHIP(n) of Lemma 2.2 on $\mu(T)$, and then the tester A. Observe that T is in D_m if and only if $\mu(T)$ is in D_1 and T is D_m-consistent, and therefore if T is in D_m it will be accepted with probability at least $2/3$.

We now prove by contrapositive that when T is 4ε-far from D_m then either $\mu(T)$ is ε-far from D_1 or T is ε-far from any D_m-consistent string. Suppose that $\mu(T)$ is ε-close to D_1 and T is ε-close to a D_m-consistent string. First note that since $\mu(T)$ is ε-close to D_1, then T contains at most $2\varepsilon n$ excess parentheses. Indeed, if T contains more than $2\varepsilon n$ excess parentheses, then we have to modify at least εn of them to obtain a string $\tilde{T}$ such that $\mu(\tilde{T}) \in D_1$, a contradiction. Since T is ε-close to a D_m-consistent string, we can modify $\leq \varepsilon n$ characters in it so that the resulting string T' is D_m-consistent. By modifying $\leq \varepsilon n$ characters of T we change its excess numbers by at most εn (see Equation 1). Therefore, the number of excess parentheses in T' is at most $3\varepsilon n$. It must be even as well. We change the first half of excess parentheses to "$(_0$", and the second half to "$)_0$", obtaining a string in D_m. □

Lemma 2.5. *Given an ε-tester A for D_2-*MEMBERSHIP$(4n)$*, one can design an $\Theta(\varepsilon)$-tester B for* TRUESTRING EQUIVALENCE(n) *with the same query complexity.*

Proof. Let $w, v \in \{0, 1, \diamond\}^n$. Define w' from w where we replace "0" by "$(_0(_0$", "1" by "$(_1(_1$", and "$\diamond$" by "$(_0)_0$", and v' from v where we replace "0" by "$)_0)_0$", "1" by "$)_1)_1$", and "$\diamond$" by "$(_0)_0$". We perform the reduction of a pair (w, v) to a string of parentheses u by concatenating w' and the reverse of v'. It is clear that this maps a pair of truestring equivalent strings to a $4n$-length string in D_2, as well as that a query to u can be simulated using a single query to w or v.

We now show that if u is ε-close to D_2, then (w, v) is $O(\varepsilon)$-close to a pair of truestring equivalent strings. It suffices to show that we can delete $O(\varepsilon n)$ characters of $T(w)$ and $T(v)$ so that the resulting strings are equal, because we can simulate a deletion from $T(w)$ or $T(v)$ by replacing the corresponding character of w or v with "$\diamond$". By definition, there is a string $\tilde{u} \in D_2$ such that the Hamming distance between u and $\tilde{u}$ is $k \leq \varepsilon \cdot (4n)$. Moreover, there is a perfect matching on the parentheses of $\tilde{u}$ such that each two matched parentheses $\tilde{u}[i], \tilde{u}[j]$ are of the same type and $|i-j+1|$ is even. We now mark some characters of $\tilde{u}$. Namely, we mark each character $\tilde{u}[i] \neq u[i]$ and its matching parenthesis. Also, if $\tilde{u}[i]$ was marked and $u[i-1, i]$ or $u[i, i+1]$ was obtained by replacing a "$\diamond$" character with the sequence "$(_0)_0$" in w or v, we mark $\tilde{u}[i-1]$ or $\tilde{u}[i+1]$ respectively, as well as its matching parenthesis in $\tilde{u}$ (some such characters might have been already marked before). Finally, we mark all untouched pairs of "$(_0)_0$" corresponding to "$\diamond$" characters.

Consider a character of $T(w)$ or $T(v)$ and the corresponding sequence $\tilde{u}[i, i+1]$. If both $\tilde{u}[i]$ and $\tilde{u}[i+1]$ are marked, we delete the character. In total, we delete $O(k) = O(\varepsilon n)$ non-"$\diamond$" characters. To show that the resulting strings are equal, note that the set of unmarked characters of $\tilde{u}$ is comprised of matching parentheses (because each time we marked a pair of matching parentheses), and contains only those characters where u and $\tilde{u}$ agree. Moreover, each unmarked character $\tilde{u}[i]$, $i \leq 2n$, is an opening parenthesis that matches some unmarked closing parenthesis $\tilde{u}[j]$, $j > 2n$, where $|j - i + 1|$ is even. □

2.2 Excess parentheses preprocessing

Parnas et al. [20] showed that it suffices to query $\tilde{O}(n^{2/3}/\varepsilon^2)$ indexes of the input string T to compute the excess numbers of any substring of T of length $\geq n^{2/3}$ with precision $\varepsilon n^{2/3}$. Below we show a new approach that will allow us to approximate excess numbers of any substring independent of its length, which is important for our recursive tester. From Equation 1 it follows that to estimate

the excess numbers it suffices to estimate the number of opening and closing parentheses in each prefix and suffix of S.

Lemma 2.6. *By querying $\tilde{O}(x^2/\Delta^2)$ indexes of a string S' of length $x \leq n$, there is an algorithm computing the number of opening or closing parentheses in any substring S of S' with precision Δ correctly with probability at least $1 - 1/n^3$.*

Proof. We query a subset of $(2x^2 \log n)/\Delta^2$ indexes of S' uniformly at random. For each substring S' of length $\leq \Delta$ we can output Δ as an approximation of the number of opening or closing parentheses. Consider now any substring S of S' of length $y \cdot \Delta$, where $1 < y \leq x/\Delta$. By Chebyshev's inequality, it contains $\geq y \cdot (x \log n/\Delta)$ queried indexes with probability $\geq 1/2$. We repeat this step $\log(2n^3)$ times to amplify the probability. As a corollary, S will contain $\geq y \cdot (x \log n/\Delta)$ queried indexes with probability $\geq 1 - 1/2n^3$. We divide the samples into $\log n$ subsets of size $y \cdot (x/\Delta)$. Consider one such subset of indexes $p_1, \ldots, p_{y\cdot(x/\Delta)}$. Setting $X_i = 1$ if $S[p_i]$ is an opening parenthesis and $X_i = 0$ otherwise, for $X = \sum_{i=1}^{y\cdot(x/\Delta)} X_i$ we have $\mathbb{E}[X] = n_0 \cdot \frac{y\cdot(x/\Delta)}{y\cdot\Delta}$. By the additive Chernoff bound we then obtain

$$\Pr\left[|X - n_0 \cdot \frac{y \cdot (x/\Delta)}{y \cdot \Delta}| \geq \sqrt{y \cdot (x/\Delta)}\right] \leq 2e^{-2} < 1/3$$

Dividing the inequality under the probability by x/Δ^2 we obtain

$$\Pr\left[|X \cdot (\Delta^2/x) - n_0| \geq \frac{\sqrt{y \cdot (x/\Delta)}}{x/\Delta^2}\right] \leq 2e^{-2} < 1/3$$

Since $\frac{\sqrt{y\cdot(x/\Delta)}}{x/\Delta^2} \leq \Delta$ (recall that $y \leq x/\Delta$), we obtain that $\hat{n}_0 = X \cdot (\Delta^2/x)$ is a Δ-approximation of n_0 with probability $> 2/3$. We amplify the probability by taking the median of the values computed over all subsets of indexes. □

Lemma 2.7. *By querying $\tilde{O}(x^2/\Delta^2)$ random indexes of a string S' of length $x \leq n$, there is an algorithm computing the excess numbers of any substring S of S' with precision Δ correctly with probability at least $1 - 1/n^3$.*

Proof. The lemma follows immediately from Equation 1 and Lemma 2.6 for $\Delta = \Delta/2$. □

2.3 Matching graph

Let us first remind the notion of a matching graph introduced by Parnas et al. [20]. Let k, ℓ be the excess numbers of T, i.e. the smallest integers such that $T' = (^k\ \mu(S)\ )^\ell \in D_1$. Since $T' \in D_1$, there is a unique perfect matching on its characters. Let $b = n^{4/5}$. We divide T into non-overlapping blocks of length b (the last block may be shorter).

Definition 2.8 (Matching graph)**.** *The matching graph $G = (V, E)$ of T is a weighted graph where V is a set of the blocks of T. If $w(i, j)$ parentheses in block i match parentheses in block j, then the two blocks i, j are connected by an edge (i, j) of weight $w(i, j)$.*

In other words, the matching graph tells if the blocks i, j contain matching parentheses, and also the number of such parentheses. Compared to the definition given in [20], we changed the size of the blocks, which will allow us to use recursion and to improve the upper bound. This change does not affect the properties of the matching graph stated in [20].

Remark 2.9 ([20]). *The matching graph G is planar and therefore has at most $3n/b$ edges.*

We say that blocks i and j are neighbours if there is an edge between them. Let $T_{i,j}$ be the substring of T containing blocks i-th to j-th inclusively.

Lemma 2.10 ([20]). *Let $i \neq j$ be two blocks of T and define $\sigma(i,j) = \min\{e_0(T_{i,i}), e_1(T_{i+1,j})\} - e_1(T_{i+1,j-1})$. The following is true: (a) If $\sigma(i,j) > 0$, then i, j are neighbours; (b) If i, j are neighbours, $w(i,j) = \sigma(i,j)$.*

We will compute the matching graph approximately using Algorithm 1. It relies on the approximation of excess parentheses with precision εb from Lemma 2.7. We call the resulting output graph the *approximate matching graph* $\hat{G}$.

Algorithm 1 Approximate matching graph $\hat{G}$

Input: string T of size n

1. Divide T into non-overlapping blocks of length $b = n^{4/5}$
2. Run the excess parentheses preprocessing for precision εb (Lemma 2.7)
3. For each $i, j \in \{1, \ldots, n/b\}$, $i \neq j$:
   (a) Get $\hat{e}_0(T_{i,i})$, $\hat{e}_1(T_{i+1,j})$, and $\hat{e}_1(T_{i+1,j-1})$
   (b) Compute $\hat{\sigma}(i,j) = \min\{\hat{e}_0(T_{i,i}), \hat{e}_1(T_{i+1,j})\} - \hat{e}_1(T_{i+1,j-1})$
4. Construct the weighted graph $\hat{G} = (V, \hat{E})$ where V is a set of blocks of T and $\hat{E}$ is the set of edges (i,j) such that $\hat{\sigma}(i,j) > 8\varepsilon b$, with respective weights $\hat{w}(i,j) = \hat{\sigma}(i,j)$

The approximate matching graph satisfies the following property.

Lemma 2.11 ([20]). *With probability at least $1 - 1/n$, the approximate matching graph $\hat{G}$ is a subgraph of the matching graph G, and every vertex in $\hat{G}$ has degree $O(1/\varepsilon)$.*

We also show a new property that will be essential for the analysis of our D_m-consistency tester. For this, define a *locally excess parenthesis* to be an excess parenthesis of some block of T which is not excess in T. We will show that the total weight of the edges of the approximate matching graph accounts for almost all locally excess parentheses.

Lemma 2.12.

$$\sum_{(i,j)\in\hat{E}:i<j} \hat{w}(i,j) \geq \frac{1}{2} \sum_{i=1}^{i=n/b} (e_0(T_{i,i}) + e_1(T_{i,i})) - (e_0(T) + e_1(T)) - O(\varepsilon n).$$

Proof. Consider an edge (i,j) of weight $w(i,j) \geq 9\varepsilon b$. We then have $\sigma(i,j) \geq 9\varepsilon b$. Consequently $\hat{\sigma}(i,j) > 8\varepsilon b$, which implies that $(i,j) \in \hat{E}$. In other words, (i,j) is an edge of $\hat{G}$ as well. By Remark 2.9, the total weight of edges $(i,j) \in E$ such that $w(i,j) < 9\varepsilon b$ is at most $3(n/b)\cdot 9\varepsilon b = 27\varepsilon n$. Therefore,

$$\sum_{(i,j)\in\hat{E}:i<j} \hat{w}(i,j) = \sum_{(i,j)\in\hat{E}:i<j} \hat{\sigma}(i,j) \geq \sum_{(i,j)\in\hat{E}:i<j} \sigma(i,j) - O(\varepsilon n) = \sum_{(i,j)\in E:i<j} w(i,j) - O(\varepsilon n)$$

Since T contains $e_0(T) + e_1(T)$ excess parentheses, we have that

$$\sum_{(i,j)\in E:i<j} w(i,j) \geq \frac{1}{2} \left(\sum_{i=1}^{i=n/b} e_0(T_{i,i}) + e_1(T_{i,i}) \right) - (e_0(T) + e_1(T))$$

Combining the two inequalities we obtain the claim. □

3 A tester for D_m-consistency

Let T be the input string partitioned into non-overlapping blocks of length $b = n^{4/5}$ (except for the last block that may be shorter). Our D_m-consistency test consists of two steps: first we check that the excess parentheses of the blocks can be matched correctly, and then (recursively) check that the blocks are D_m-consistent. The structure of the test repeats the structure of the test by Parnas et al. [20], in particular, both tests are based on the notion of approximate matching graph. However, our test uses new and much more sophisticated techniques, which finally gives us a better bound.

Algorithm 2 D_m-CONSISTENCY(n, k)

Input: string T of length n

1. If $k = 0$, run the deterministic D_m-consistency 0-tester, and stop
2. Divide T into non-overlapping blocks of length $b = n^{4/5}$
3. Inter-block matching:
   (a) Compute the approximate matching graph $\hat{G}$ for the blocks using Algorithm 1
   (b) Select $\varepsilon^{-1} \log n$ blocks uniformly at random and for each find all of its $O(1/\varepsilon)$ neighbours in $\hat{G}$
   (c) For each selected block i and for each of its neighbours j:
      i. Find (approximately) the smallest substring S_1 of block i that contains all excess opening parentheses that match in block j, and the smallest substring S_2 of block j that contains all excess closing parentheses that match in block i (see Section 3.1)
      ii. Check that S_1, S_2 ε-match using SUBSTRING ε-MATCHING(b) (Theorem 3.2)
4. D_m-consistency of the blocks:
   (a) Select $4\varepsilon^{-1} \log n$ blocks uniformly at random
   (b) Run the D_m-CONSISTENCY$(b, k-1)$ test twice for each of the selected blocks

Algorithm 2 shows the main steps of our new tester D_m-CONSISTENCY(n, k). In Step 3 we call a subroutine SUBSTRING ε-MATCHING(x). It must accept S_1 if almost all excess parentheses in it can be matched in S_2. Formally,

Definition 3.1 (sequentially match, ε-match)**.** *Consider two substrings S_1, S_2 of T of maximal length x. We say that the excess opening parentheses of S_1 can be* matched sequentially *in S_2 if there is a perfect matching between the excess opening parentheses of S_1 and a (continuous) subrange of excess closing parentheses of S_2 such that any two matched parentheses $T[i], T[j]$ have the same type and the* distance between them*, defined as $|j - i + 1|$, is even.*

We say that S_1, S_2 ε-match *if all but at most $7\varepsilon x$ leftmost and $5\varepsilon x$ rightmost excess opening parentheses of S_1 can be matched sequentially in S_2 .*

SUBSTRING ε-MATCHING(x)
Input: Two substrings S_1, S_2 of T of maximal size x
Output: Accept if they ε-match; Reject if at most $e_0(S_1) - 30\varepsilon x$ excess opening parentheses of S_1 can be matched sequentially in S_2.

The choice of constants is important for the analysis of D_m-CONSISTENCY(n, k). In Section 4 we show the following theorem by using recursion.

Theorem 3.2. *For every constant $0 < \delta \leq 1/2$ there is an algorithm for* Substring ε-matching(x) *with query complexity $\tilde{O}(\varepsilon^{(2\delta)^{(1/\log 3/4)-2}+4} x^{1/2+\delta})$ and bounded error $1/n^2$.*

In Section 3.1 we give a detailed description of Algorithm 2, Step 3 (inter-block matching), and then in Section 3.2 prove the following theorem.

Theorem 3.3. *For any $0 < \delta < 1/2$, D_m-*consistency$(n, 5)$ *(Algorithm 2) is an ε-tester for D_m-*consistency(n) *with query complexity $\tilde{O}(\varepsilon^{(2\delta)^{(1/\log 3/4)-2}+2} n^{2/5+4/5\cdot\delta})$ and bounded error $1/6$.*

This theorem immediately implies a new tester for D_m-membership via Lemma 2.4.

Corollary 3.4. *For any $0 < \delta < 1/2$ there is an ε-tester for D_m-*membership(n) *with query complexity $\tilde{O}(\varepsilon^{(2\delta)^{(1/\log 3/4)-2}+2} n^{2/5+4/5\cdot\delta})$.*

3.1 Inter-block matching

In this section we give a detailed description and analyse Step 3 of the D_m-consistency(n, k) algorithm (inter-block matching). We start by running the excess parentheses preprocessing (Lemma 2.7) and computing the approximate matching graph for the blocks. Next, we select $\varepsilon^{-1} \log n$ blocks uniformly at random and for each find all of its $O(1/\varepsilon)$ neighbours in the approximate matching graph.

Consider one of the selected blocks, i and its neighbour j. Assume $i < j$, the other case is symmetrical. Let $[p, q]$ be the smallest interval of indexes in the block i containing all excess opening parentheses that match in the block j, and $[r, s]$ the smallest interval of indexes in the block j containing all closing parentheses that match in block i. Unfortunately, we cannot compute the precise values of p, q, r, s, but we can compute their approximate values $\hat{p}, \hat{q}, \hat{r}, \hat{s}$ (see Figure 1). We run the Substring ε-matching algorithm (Theorem 3.2) on $T[\hat{p}, \hat{q}]$ and $T[\hat{r}, \hat{s}]$. If the algorithm confirms that the substrings ε-match, we accept i and j, and otherwise we reject them. The inter-block matching accepts T if and only if all tested block pairs are accepted.

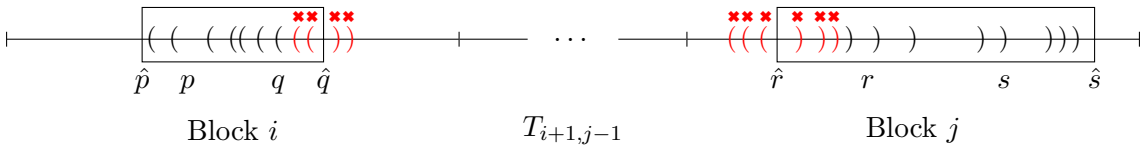


Figure 1: Blocks i and j, and intervals $[\hat{p}, \hat{q}]$ and $[\hat{r}, \hat{s}]$. Black parentheses are excess parentheses of the blocks i and j, red parentheses (also marked by crosses ✖) are excess in $T[\hat{p}, \hat{q}]$ and $T[\hat{r}, \hat{s}]$, but not in the blocks i, j (they will be matched with the red parentheses just outside of the intervals.)

We now explain how we approximate p, q, r, s. Recall that $\hat{w}(i, j) > 8\varepsilon b$ is the approximate weight of the edge (i, j) and $\hat{e}_0(S)$, $\hat{e}_1(S)$ are the excess numbers of opening / closing parentheses in a substring S of T computed with precision εb. We compute $\hat{p}, \hat{q}, \hat{r}, \hat{s}$ in the following way:

1. Let $\hat{q}$ be the rightmost index in block i such that $\hat{e}_0(T[\hat{q}, ib]) \geq \hat{e}_1(T_{i+1,j-1}) - 2\varepsilon b$ and $\hat{e}_1(T[\hat{q}, ib]) \leq \varepsilon b$;
2. Let $\hat{p}$ be the rightmost index in block i such that $\hat{e}_0(T[\hat{p}, \hat{q}]) \geq \hat{w}(i, j) + 6\varepsilon b$ (if there is none, we put $\hat{p} = (i-1)b + 1$);
3. Let $\hat{r}$ be the leftmost index in block j such that $\hat{e}_1(T[(j-1)b+1, \hat{r}]) \geq \hat{e}_0(T_{i+1,j-1}) - 2\varepsilon b$ and $\hat{e}_0(T[(j-1)b+1, \hat{r}]) \leq \varepsilon b$;

4. Let $\hat{s}$ be the leftmost index in block j such that $\hat{e}_1(T[\hat{r},\hat{s}]) \geq \hat{w}(i,j) + 6\varepsilon b$ (if there is none, we put $\hat{s} = jb$).

Since $\hat{w}(i,j) = \hat{\sigma}(i,j) = \min\{\hat{e}_0(T_{i,i}), \hat{e}_1(T_{i+1,j})\} - \hat{e}_1(T_{i+1,j-1}) > 8\varepsilon b$, indexes $\hat{q}$ and $\hat{r}$ always exist.

Correctness of inter-block matching. We now show that if T is D_m-consistent, inter-block matching will accept it with high probability, and that if T is accepted, then almost all locally excess parentheses of T can be matched correctly. Recall that a parenthesis of T is called locally excess if it is excess for some block of T, but not for T itself.

Lemma 3.5. *If T is D_m-consistent, it is accepted by inter-block matching with probability $> 1-1/n$.*

Proof. We will show that if T is D_m-consistent, then for every pair of neighbours i, j in $\hat{G}$ the substrings $T[\hat{p},\hat{q}]$ and $T[\hat{r},\hat{s}]$ defined as above ε-match. From Theorem 3.2 and the union bound it will immediately follow that T is accepted with probability $> 1 - 1/n$.

We need to show that all but at most $7\varepsilon b$ leftmost and at most $5\varepsilon b$ rightmost excess opening parentheses in $T[\hat{p},\hat{q}]$ can be sequentially matched in $T[\hat{r},\hat{s}]$. By definition, all excess parentheses of $T[p,q]$ can be sequentially matched in $T[r,s]$. $T[\hat{r},\hat{s}]$ contains $T[r,s]$ as a subinterval, and therefore it suffices to show that $T[\hat{p},\hat{q}]$ has at most $5\varepsilon b$ extra excess opening parentheses on the right and at most $7\varepsilon b$ extra excess opening parentheses on the left compared to $T[p,q]$.

If T is D_m-consistent, all excess closing parentheses in $T_{i+1,j-1}$ must match in $T[q+1, ib]$. Therefore, $T[q, ib]$ must contain $e_1(T_{i+1,j-1})$ excess opening parentheses. From the definition it follows that $T[\hat{q}, ib]$ contains at least $e_1(T_{i+1,j-1}) - 3\varepsilon b$ parentheses. It means that $T[\hat{p},\hat{q}]$ has at most $3\varepsilon b$ extra excess opening parentheses on the right that must be matched in $T_{i+1,j-1}$. There also can be at most $2\varepsilon b$ extra excess opening parentheses that were not excess in the block i but became excess in $T[\hat{p},\hat{q}]$ (see Figure 1). In total, there will be at most $5\varepsilon b$ extra excess opening parentheses on the right.

Consider now the rightmost excess opening parenthesis of $T[\hat{p},\hat{q}]$ that matches in block j. Starting from it, there must be $w(i,j)$ more excess opening parentheses that also match in block j. We defined $\hat{p}$ to be the rightmost index in block i such that $\hat{e}_0(T[\hat{p},\hat{q}]) \geq \hat{w}(i,j) + 6\varepsilon b$. It follows that $\hat{e}_0(T[\hat{p}+1,\hat{q}]) < \hat{w}(i,j) + 6\varepsilon b < w(i,j) + 7\varepsilon b$ or that $e_0(T[\hat{p},\hat{q}]) \leq w(i,j) + 7\varepsilon b$, which concludes the proof. □

Lemma 3.6. *If T is accepted with probability $> 1/n$, then there is a matching on its locally excess parentheses such that: (a) Any two matched parentheses have the same type and the distance between them is even; (b) There are at most $O(\varepsilon n)$ unmatched locally excess parentheses.*

Proof. By Lemma 2.7, excess preprocessing is correct for all substrings of T with probability $> 1 - 1/2n$. We can therefore assume that both assumptions (T is accepted, excess preprocessing is correct) hold with probability $> 1/2n$.

The fraction of blocks i, which have a neighbour j in $\hat{G}$ such that the substring matching test accepts with probability $< 1/n^2$ if executed on i, j, is at most ε (otherwise we would select one of them with probability $\geq 1 - 1/n^2$). Let $\mathcal{R}$ be a set of all such blocks. We thus obtain that

$$\sum_{(i,j)\in\hat{E}: i\neq j, i\in\mathcal{R}} \hat{\sigma}(i,j) = O(\varepsilon n)$$

Consider now any three blocks $i_1 < i_2 < j$ (the case $j < i_1 < i_2$ is analogous.) such that both i_1 and i_2 are neighbours of j in $\hat{G}$. Suppose that both i_1, j and i_2, j are accepted by the SUBSTRING ε-MATCHING algorithm with probability $> 1/n^2$. Theorem 3.2 implies that there is a subsequence

of locally excess opening parentheses of block i_1 of length $\geq \sigma(i_1, j) - 30\varepsilon b$ that can be matched with a subsequence π_1 of locally excess closing parentheses of block j, and a subsequence of locally excess opening parentheses of block $i_2 \geq \sigma(i_1, j) - 30\varepsilon b$ that can be matched with a subsequence π_2 of locally excess closing parentheses of block j. The matchings satisfy property (a); moreover, assuming that the excess numbers of all substrings of T were computed correctly with precision εb, the subsequences π_1 and π_2 overlap by $\leq 12\varepsilon b$ excess parentheses. The latter follows from the definition of $\hat{p}, \hat{q}, \hat{r}, \hat{s}$. Indeed, the number of excess closing parentheses between $(j-1)b$ and the last parenthesis of π_2 is at most $\hat{e}_0(T_{i_2+1,j-1}) + \hat{w}(i_2, j) + 7\varepsilon b \leq e_0(T_{i_2+1,j-1}) + \sigma(i_2, j) + 9 \cdot \varepsilon b$, where $\sigma(i_2, j) = \min\{e_0(T_{i_2,i_2}), e_1(T_{i_2+1,j})\} - e_1(T_{i_2+1,j-1}) \leq e_0(T_{i_2,i_2}) - e_1(T_{i_2+1,j-1})$. On the other hand, the number of excess closing parentheses between $(j-1)b$ and the first parenthesis of π_1 is at least $e_0(T_{i_1,j-1}) - 3\varepsilon b$. Note that $e_0(T_{i_1,j-1}) \geq e_0(T_{i_2,i_2}) - e_1(T_{i_2+1,j-1}) + e_0(T_{i_2+1,j-1})$. Therefore, the two subsequences overlap by at most $12\varepsilon b$ excess parentheses.

From above it follows that the total number of locally excess parentheses in the matched subsequences is $\sum_{(i,j)\in\hat{E}:i<j} \hat{\sigma}(i, j) - O(\varepsilon n)$. Moreover, each two subsequences overlap by at most $12\varepsilon b$ parentheses. Since by Lemma 2.11 $\hat{G}$ is a subgraph of G, that is planar and therefore has at most $3n/b$ edges, the total lengths of overlaps is $O(\varepsilon n)$. Therefore, we will be able to match at least $\sum_{(i,j)\in\hat{E}:i<j} \hat{\sigma}(i, j) - O(\varepsilon n)$ locally excess parentheses. The claim follows from Lemma 2.12. □

Complexity of inter-block matching. To compute the approximate matching graph we need $\tilde{O}(\varepsilon^{-2} n^{2/5})$ queries. The substring matching test is called $\tilde{O}(\varepsilon^{-2} \log n)$ times, and takes $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4-2}+4} b^{1/2+\delta})$ queries for a fixed constant $0 < \delta < 1/2$. Since $b = n^{4/5}$, we finally obtain that inter-block matching can be implemented to have complexity $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4-2}+2} n^{2/5+4/5\cdot\delta})$ for any $0 < \delta < 1/2$.

3.2 Recursion (proof of Theorem 3.3)

We are now ready to prove Theorem 3.3 that claims that D_m-CONSISTENCY$(n, 5)$ test is an ε-tester for D_m-CONSISTENCY(n) with complexity $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4-2}+2} n^{2/5+4/5\cdot\delta})$.

We start by analysing the complexity of the test. The pseudocode of the test (see Algorithm 2) directly implies that if D_m-CONSISTENCY$(n, k-1)$ test has query complexity $f(n, \varepsilon)$, then D_m-CONSISTENCY(n, k) test has query complexity $\tilde{O}(\varepsilon^{-1} f(n^{4/5}, \varepsilon) + \varepsilon^{(2\delta)^{1/\log 3/4-2}+2} n^{2/5+4/5\cdot\delta})$, where $0 < \delta < 1/2$ is a constant in the complexity of SUBSTRING ε-MATCHING (Theorem 3.2). Recall that for the base case $k = 0$, the D_m-CONSISTENCY$(n, 0)$ test is the trivial deterministic 0-tester for D_m-consistency with query complexity $f(n, \varepsilon) = n$. Therefore after applying the recursive step five times, we obtain a test with complexity $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4-2}+2} n^{2/5+4/5\cdot\delta})$.

Correctness. We now show that the D_m-CONSISTENCY$(n, 5)$ test is an ε-tester for D_m-CONSISTENCY(n) with bounded error $1/6$. By the definition, we need to show that if T is D_m-consistent, the test will accept it with probability $> 1 - 1/6$, and if T is accepted with probability $> 1/6$, T is $\mathbf{C} \cdot \varepsilon$-close to D_m-consistent for some constant $\mathbf{C} > 0$.

We start with the first part of the claim. Suppose that if T is D_m-consistent, then it is accepted by the D_m-CONSISTENCY$(n, k-1)$ test with probability $> 1-\alpha$. By the union bound and Lemma 4.2 it then follows that D_m-CONSISTENCY(n, k) test will accept T with probability $> 1 - \alpha^2 \cdot \varepsilon^{-1} \log n - 1/n$ (the test can err either in the inter-block matching, or in one of the $\varepsilon^{-1} \log n$ calls to D_m-CONSISTENCY$(n, k-1)$). Since for $k = 0$ the error probability $\alpha = 0$, we obtain that the error probability for $k = 5$ is less than $1/6$.

We now show the second part of the claim by induction on k. Suppose that the following is true: If the D_m-CONSISTENCY$(n, k-1)$ test accepts T with probability $> 1/6$, then it is $\mathbf{C}_{k-1} \cdot \varepsilon$-close to D_m-consistent for some constant $\mathbf{C}_{k-1} > 0$. We will now show that if the D_m-CONSISTENCY(n, k) test accepts T with probability $> 1/6$, then it is $\mathbf{C}_k \cdot \varepsilon$-close to D_m-consistent for some constant $\mathbf{C}_k > 0$. This will conclude the proof of Theorem 3.3. We do this in three steps as described below: First, we show how to make all the blocks D_m-consistent, secondly, we show that we can match almost all locally excess parentheses in the resulting string, and finally, we show how to modify the remaining locally excess parentheses so that we can match them as well.

Step 1 - Making all blocks D_m-consistent. Since the D_m-CONSISTENCY(n, k) test accepts T with probability $> 1/6$, at most ε-fraction of the blocks can be accepted by the D_m-CONSISTENCY$(n, k-1)$ test with probability $\leq 1/6$. Indeed, if there were more than ε-fraction of such blocks, one of them would be selected with probability $> 1 - 1/n$. Recall that we run the D_m-CONSISTENCY$(n, k-1)$ test on this block twice (Algorithm 2, Step 4(b)) . It means that it will be accepted by the test with probability $\leq 1/36$. By the union bound we obtain that in this case T would be accepted with probability $< 1/6$, a contradiction. It follows that at least a $(1-\varepsilon)$-fraction of the blocks would be accepted by the D_m-CONSISTENCY$(n, k-1)$ test with probability $> 1/6$, and by our assumption they are $\mathbf{C}_{k-1} \cdot \varepsilon$-close to D_m-consistent.

We now explain how we modify the blocks to make them D_m-consistent. First we consider all blocks that are $\mathbf{C}_{k-1} \cdot \varepsilon$-far from D_m-consistent. For every such block B, there is a unique perfect matching on the non-excess parentheses. If a pair of matched non-excess parentheses have different types, we modify one of them accordingly. Note that this procedure does not change the set of excess parentheses of such blocks. In total, we modify at most $\varepsilon n/2$ parentheses (up to $b/2$ in each such block). We now consider the blocks that are $\mathbf{C}_{k-1} \cdot \varepsilon$-close to D_m-consistent. By definition we can make each such block D_m-consistent by modifying $\leq \mathbf{C}_{k-1} \cdot \varepsilon b$ parentheses in it. In total, we modify at most $\mathbf{C}_{k-1} \cdot \varepsilon n$ parentheses. We denote the resulting string by T'. From Equation 1 we immediately obtain the following observation, that will be important for further analysis.

Observation 3.7. *After modifying $\leq \mathbf{C}_{k-1} \cdot \varepsilon b$ characters of a block, the set of excess opening/closing parentheses in it can change by at most $2\mathbf{C}_{k-1} \cdot \varepsilon b$ parentheses.*

We finally obtain that the sets of excess parentheses of T and T' differ by at most $2\mathbf{C}_{k-1} \cdot \varepsilon n$ parentheses.

Step 2 - Partial matching of locally excess parentheses. We now build a matching on the locally excess parentheses of T'. We first consider the initial non-modified string T. The inter-block matching test must accept T with probability $> 1/6$ and therefore by Lemma 4.2 there is a matching on locally excess parentheses of T such that: (a) Any two matched parentheses have the same type and the distance between them is even; (b) There are at most $O(\varepsilon n)$ unmatched locally excess parentheses. We now consider an induced matching on excess parentheses of T'. Namely, we match two locally excess parentheses of T' if they are matched in T and if they were not modified during the first step. From Lemma 3.6 it follows that T' will contain at most $O(\varepsilon n)$ non-matched locally excess parentheses.

Step 3 - Modifying non-matched excess parentheses. The string T' is now composed of four types of consecutive substrings: (a) Substrings that belong to D_m; (b) Locally excess parentheses in one block that are matched with locally excess parentheses in another block; (c) At most $O(\varepsilon n)$ locally excess parentheses that are not matched (see Step 2); and (d) At most $O(\varepsilon n)$

excess parentheses of T' that are not excess parentheses of T (see Step 1); (e) Excess parentheses of T.

Let T'' be the string obtained from T' by removing all substrings of type (a). Note that by removing such substrings we do not change the parity of the distance between any two matched excess parentheses. We show how to modify T'' in a recursive way. Let t', t'' be two matched substrings of excess parentheses such that between them there is only one substring τ of parentheses of types (c) or (d). It follows that $S'' = p't'\tau t''p''$ for some strings p' and p''. The length of τ is even. We replace it with an arbitrary string in D_m, and then remove $t'\tau t''$ from S'' and continue recursively. In the end we will obtain a set of excess parentheses of T. This concludes the proof of Theorem 3.3.

4 Algorithm for Substring ε-matching

In this section we show an algorithm for Substring ε-matching with bounded error $1/3$ and query complexity $\tilde{O}(\varepsilon^{2\delta^{1/\log 3/4-2}+4}x^{1/2+\delta})$. Theorem 3.2 will immediately follow, as we can repeat the algorithm a logarithmic number of times to boost the probability.

4.1 Algorithm for Substring ε-matching with bounded error $1/3$

Recall that the algorithm receives as an input two substrings S_1, S_2 of T of length at most x, and must accept S_1, S_2 if they ε-match, and reject if at most $e_0(S_1) - 30\varepsilon x$ excess opening parentheses of S_1 can be sequentially matched in S_2. The algorithm consists of three recursive procedures: Procedures QueryLeft(x, ε, k) and QueryRight(x, ε, k) query a subset of characters of strings S_1 and S_2, and the third procedure, MakeDecision(x, ε, k) accepts or rejects (S_1, S_2) using the queried characters. We give the pseudocode of our solution in Algorithm 3.

Algorithm 3 Substring ε-matching

Input: Two substrings S_1, S_2 of a string T of length $\leq x$

1. $k := \lceil \log_{3/4} 2\delta \rceil$
2. Run QueryLeft(x, ε, k) for S_1
3. Run QueryRight(x, ε, k) for S_2
4. MakeDecision(x, ε, k)

We now describe each procedure in turn.

Procedures QueryLeft(x, ε, k) and QueryRight(x, ε, k). Let $x' = x^{3/4}$ and $\varepsilon' = \varepsilon/30$. Procedure QueryLeft$(x, \varepsilon, k)$ starts by running the excess numbers preprocessing on S_1 for precision $(\varepsilon')^2 x'$. Next, it partitions S_1 into non-overlapping blocks starting from the right. If the approximate number of excess parentheses in S_1 is less than $10\varepsilon' x'$, the partitioning of S_1 is defined to contain a single block equal to S_1 itself. Otherwise, it must satisfy the following two properties: (1) The approximate number of excess opening parentheses in the m rightmost blocks of S_1 is between $(10m - \varepsilon') \cdot \varepsilon' x'$ and $(10m + \varepsilon') \cdot \varepsilon' x'$; and (2) The approximate number of excess closing parentheses in these blocks is at most $(\varepsilon')^2 x'$. (Note that such a partitioning always exists because, for example, we can choose the m-th block to be the substring bounded by the $10(m-1)$-th and $10m$-th excess parentheses. The procedure might choose another partitioning, but this shows that it will have at least one possible choice.) We call blocks of length $\leq x'$ *dense*. Finally, the procedure deletes the leftmost and the rightmost blocks of S_1.

Procedure 1 QUERYLEFT(x, ε, k)

Input: Substring S_1 of T of length $\leq x$

Output: Partitioning of S_1, a sequence $\mathcal{L} = \{\mathcal{L}_t\}$ of subsets of dense blocks, queried characters

1. If $k = 0$, query all characters of S_1
2. $x' := x^{3/4}$, $\varepsilon' := \varepsilon/30$
3. Run excess numbers preprocessing for S_1 with precision $(\varepsilon')^2 x'$
4. Partition S_1 into blocks containing approximately $10\varepsilon' x'$ excess opening parentheses, and then delete the leftmost and the rightmost blocks
5. Select a sequence $\mathcal{L} = \{\mathcal{L}_t\}$ of $\tilde{\Theta}(\varepsilon^{-1})$ random subsets of dense blocks of S_1 of size $\tilde{\Theta}((\varepsilon')^{-1}\sqrt{x/x'})$
6. Run QUERYLEFT$(x', 2(\varepsilon')^2, k-1)$ for each selected block $\Theta(\log(x \cdot \varepsilon^{-1} \log x))$ times

Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be positive constants to be defined later. The procedure selects $\mathbf{A} \cdot \varepsilon^{-1} \log x$ random subsets $\mathcal{L}_t$ of dense blocks of S_1, where each subset has size $\mathbf{B} \cdot (\varepsilon')^{-1}\sqrt{x/x'} \log(x \cdot \varepsilon^{-1} \log x)$. For each selected dense block B the procedure runs QUERYLEFT$(x', 2(\varepsilon')^2, k-1)$ independently $\mathbf{C} \cdot \log(x \cdot \varepsilon^{-1} \log x)$ times over B. The pseudocode is given in Procedure 1.

Similar to above, procedure QUERYRIGHT(x, ε, k) starts by running the excess numbers preprocessing on S_2 for precision $(\varepsilon')^2 x'$. For technical reasons that will become clear later, we consider not just one partitioning of S_2, but a number of them. Namely, we consider a separate partitioning for each shift $\tau = (\varepsilon')^2 x', 2(\varepsilon')^2 x', \ldots, 12 \cdot \varepsilon x'$ (in total, we have $12\varepsilon/(\varepsilon')^2 = 12 \cdot 30/\varepsilon'$ shifts). For each $m \geq 0$ the approximate number of excess closing parentheses in the $(m+1)$ leftmost blocks of the partitioning must be between $\tau + (10m - \varepsilon') \cdot \varepsilon' x'$ and $\tau + (10m + \varepsilon') \cdot \varepsilon' x'$, and the approximate number of excess opening parentheses must be at most $(\varepsilon')^2 x'$.

Procedure 2 QUERYRIGHT(x, ε, k)

Input: Substring S_2 of T of length $\leq x$

Output: Partitionings of S_2, sequences $\mathcal{R}(\tau) = \{\mathcal{R}_t(\tau)\}$ of subsets of dense blocks, queried characters

1. If $k = 0$, query all characters of S_2
2. $x' := x^{3/4}$, $\varepsilon' := \varepsilon/30$
3. Run excess numbers preprocessing for S_2 with precision $(\varepsilon')^2 x'$
4. For each shift $\tau \in \{(\varepsilon')^2 x, 2(\varepsilon')^2 x, \ldots, 12\varepsilon x\}$:
   (a) Partition S_2 into blocks containing approximately $10\varepsilon' x'$ excess closing parentheses, except for the first block containing approximately τ excess closing parentheses
   (b) Select a sequence $\mathcal{R}(\tau) = \{\mathcal{R}_t(\tau)\}$ of $\tilde{\Theta}(\varepsilon^{-1})$ sets of dense blocks of S_2 of size $\tilde{\Theta}((\varepsilon')^{-1}\sqrt{x/x'})$
   (c) Run QUERYRIGHT$(x', 2(\varepsilon')^2, k-1)$ for each selected block $\Theta(\log(x \cdot \varepsilon^{-1} \log x))$ times

For the partitioning of S_2 corresponding to a shift τ, the procedure selects $\mathbf{A} \cdot \varepsilon^{-1} \log x$ random subsets $\mathcal{R}_t(\tau)$ of dense blocks of S_2 of size $\mathbf{B} \cdot (\varepsilon')^{-1}\sqrt{x/x'} \log(x \cdot \varepsilon^{-1} \log x)$ each. For each selected dense block B the procedure runs QUERYRIGHT$(x', 2(\varepsilon')^2, k-1)$ independently $\mathbf{C} \cdot \log(x \cdot \varepsilon^{-1} \log x)$ times. The pseudocode is given in Procedure 2.

Procedure MakeDecision(ε, x, k)**.** We finally explain how we use the queried indexes to test S_1 and S_2. If $k = 0$, QueryLeft(ε, x, k) and QueryRight(ε, x, k) know all characters of S_1 and S_2 and we can use a naive deterministic algorithm to decide whether S_1 and S_2 ε-match. If $\hat{e}_0(S_1) < 10\varepsilon x$, we always accept S_1 and S_2.

Suppose now that $k > 0$ and $\hat{e}_0(S_1) \geq 10\varepsilon x$. For each partitioning of S_2 we consider all possible substrings X that start at some block border. We process each of them in turn and accept (S_1, S_2) if at least one of the substrings X is accepted. If the difference between the approximate number of excess opening parentheses in S_1, $\hat{e}_0(S_1)$, and excess closing parentheses in X, $\hat{e}_1(X)$ is larger than $4(\varepsilon')^2 x'$, X is rejected, and otherwise we continue to the next step. X is tested in $\mathbf{A} \cdot \varepsilon^{-1} \log x$ iterations, and we accept X if and only if it is accepted at each iteration. Suppose that X starts at a block border of a partitioning for a shift τ. We enumerate the blocks in S_1 from right to left and the blocks in X from left to right. At iteration $t = 1, 2, \ldots, \mathbf{A} \cdot \varepsilon^{-1} \log x$ we find a rank m such that the m-th block B_1 of S_1 is in the subset $\mathcal{L}_t$, and the m-th block B_2 of S_2 is in $\mathcal{R}_t(\tau)$. Finally, we run the MakeDecision$(x', 2(\varepsilon')^2, k-1)$ procedure on (B_1, B_2) independently $\mathbf{C} \cdot \log(x \cdot \varepsilon^{-1})$ times; and if the blocks are rejected for the majority of iterations, reject X. (See Procedure 3.)

Procedure 3 MakeDecision(x, ε, k)

Input: Substrings S_1, S_2 of a string T of length $\leq x$; outputs of QueryLeft(x, ε, k) run on S_1 and of QueryRight(x, ε, k) run on S_2

1. If $k = 0$, use the trivial algorithm to decide whether S_1 and S_2 ε-match
2. $x' := x^{3/4}$, $\varepsilon' := \varepsilon/30$
3. If $\hat{e}_0(S_1) < 10\varepsilon x$, accept S_1, S_2
4. For each partition of S_2 and for each substring X starting at the partition's block border:
   (a) If $|\hat{e}_0(S_1) - \hat{e}_1(X)| > 4(\varepsilon')^2 x'$, reject X
   (b) Find $\Theta(\varepsilon^{-1} \log x)$ pairs of dense blocks of S_1 and X that have equal ranks using sets $\mathcal{L}_t$ and $\mathcal{R}_t(\tau)$. For each such pair (B_1, B_2):
      i. Run MakeDecision$(x', 2(\varepsilon')^2, k-1)$ on (B_1, B_2) $\Theta(\log(\varepsilon^{-1} x \log x))$ times
      ii. If (B_1, B_2) is rejected for the majority of iterations, reject X
   (c) Accept (S_1, S_2) if X is not rejected

4.2 Analysis (proof of Theorem 3.2)

We now show complexity and correctness of the algorithm.

Lemma 4.1. *The query complexity of Algorithm 3 is* $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4 - 2} + 4} x^{1/2+\delta})$.

Proof. Since the algorithm queries S_1 and S_2 only during the procedures QueryLeft(x, ε, k) and QueryRight(x, ε, k), it suffices to estimate their query complexity. Let us first analyse one recursive step. The description of the procedures implies that if the query complexities of QueryLeft$(x, \varepsilon, k-1)$ and QueryRight$(x, \varepsilon, k-1)$ are bounded by $f(x, \varepsilon)$, then the sum of query complexities of the procedures is $\tilde{O}(\varepsilon^{-4}\sqrt{x} + f(x^{3/4}, 2(\varepsilon/30)^2) \cdot \varepsilon^{-3} x^{1/8} \log^2 \varepsilon^{-1} x)$. Therefore, if $f(x, \varepsilon) = \tilde{O}(\varepsilon^y x^{1/2+z})$, then the sum of query complexities of QueryLeft(x, ε, k) and QueryRight(x, ε, k) is $\tilde{O}(\varepsilon^{2y-4} x^{1/2+3/4 \cdot z})$ (we use $\log^2 \varepsilon^{-1} < \varepsilon^{-1}$ and omit all $\log x < \log n$ terms). After r iterations the degree of x becomes $1/2 + (3/4)^r \cdot y$, and the degree of ε becomes $2^r(z-4)+4$.

Recall that the query complexity of the trivial algorithm for $k = 0$ is $f(x, \varepsilon) = x = \varepsilon^y x^{1/2+z}$, where $y = 0$ and $z = 1/2$. Therefore, after $r = \log_{3/4} 2\delta$ recursive steps we obtain an algorithm with query complexity $\tilde{O}(\varepsilon^{(2\delta)^{1/\log 3/4 - 2} + 4} x^{1/2+\delta})$. □

We now show that Algorithm 3 has bounded error 1/3.

Lemma 4.2. *If S_1, S_2 ε-match, Algorithm 3 accepts them with probability $> 2/3$.*

Proof. We will show that if MakeDecision$(\varepsilon, x, k-1)$ accepts ε-matching strings with probability $> 2/3$, then MakeDecision(ε, x, k) accepts ε-matching strings with probability $> 2/3$ as well. The claim will then follow by induction, as MakeDecision$(\varepsilon, x, 0)$ (the trivial algorithm) accepts ε-matching strings with probability 1.

By the definition of ε-matching, all but at most $7\varepsilon x$ leftmost and $5\varepsilon x$ rightmost excess opening parentheses of S_1 can be sequentially matched in S_2. Assume that all excess numbers are approximated correctly, which is true with probability $> 1 - 1/2n$ and consider a subsequence π of excess opening parentheses in S_1 that contains all excess opening parentheses of S_1 except for those that belong to its leftmost and rightmost blocks. We rank the parentheses in π from right to left. Let p be the leftmost excess parenthesis in S_2 matched with a parenthesis of π. We rank the excess closing parentheses of S_2 from left to right, starting from p. Across all partitions of S_2, take the rightmost block border preceding p and let X be a substring of S_2 starting at it. We will show that X is accepted by the algorithm with probability $> 2/3$, from which the claim follows. Note that the number of excess parentheses of S_2 between the left-endpoint of X and p is at most $3(\varepsilon')^2 x'$.

Recall that MakeDecision(x, ε, k) finds a subset of dense (that is, of length $\leq x'$) blocks of S_1 and X that have equal ranks and runs MakeDecision$(x', 2(\varepsilon')^2, k-1)$ on them independently $\mathbf{C} \cdot \log(x \cdot \varepsilon^{-1})$ times. Consider the m-th block B_1 of S_1 (with the rightmost block deleted) and the m-th block B_2 of X. We will show that all but at most $14(\varepsilon')^2 x'$ leftmost and $10(\varepsilon')^2 x'$ rightmost excess parentheses of B_1 can be sequentially matched in B_2, which means that B_1, B_2 $2(\varepsilon')^2$-match. The rank of the rightmost excess opening parenthesis in B_1 is at least $(10m - 2\varepsilon')\varepsilon' x'$. The rank of the leftmost excess opening parenthesis in B_1 is at most $(10(m+1)+2\varepsilon')\varepsilon' x'$. Also, B_1 can end with at most $2(\varepsilon')^2 x'$ excess opening parentheses that are not excess parentheses of S_1. The rank of the leftmost excess closing parenthesis in B_2 is at most $(10m+2\varepsilon')\varepsilon' x'$. The rank of the rightmost excess closing parenthesis in B_2 is at least $(10(m+1) - 5\varepsilon')\varepsilon' x'$. Consequently, all but at most $7(\varepsilon')^2 x' < 14(\varepsilon')^2 x'$ leftmost excess parentheses and $6(\varepsilon')^2 x' < 10(\varepsilon')^2 x$ rightmost excess parentheses of x can be matched in B_2 as required. By our assumption, MakeDecision$(x', 2(\varepsilon')^2, k-1)$ accepts (B_1, B_2) with probability $> 2/3$, and therefore we can select the constant $\mathbf{C}$ so that B_1 and B_2 are accepted with probability at least $1 - 1/(6\mathbf{A}\varepsilon^{-1} \log x)$. From the union bound it follows that all of the $\mathbf{A} \cdot \varepsilon^{-1} \log x$ pairs of blocks for which we run the recursive call will be accepted with probability at least $1 - 1/6$, and consequently S_1 and X will be accepted with probability $> 2/3$. □

We now show by contrapositive that if at most $e_0(S_1) - 30\varepsilon x$ excess opening parentheses of S_1 can be matched sequentially in S_2, S_1 and S_2 will be rejected with probability $> 2/3$. We start with an auxiliary lemma. Recall that at each iteration t the procedure MakeDecision(x, ε, k) considers a block partitioning of S_2 with some shift τ and chooses a subset $\mathcal{L}_t$ of blocks of S_1 and a subset $\mathcal{R}_t(\tau)$ of blocks of the partition of S_2. Using these two subsets, it tests S_1 and each substring X of S_2 that starts at a block border of the partition of S_2. We rank the blocks in X from left to right. Blocks of S_1 are ranked in the reverse order, from right to left. Intuitively, two blocks of S_1 and X have equal ranks if they contain many parentheses that must be matched, and therefore we can recurse on them. Below we show that $\mathcal{L}_t$ and $\mathcal{R}_t(\tau)$ contain such blocks with high probability.

Lemma 4.3. *Assume that the excess numbers preprocessing for S_1 and S_2 did not error and that $\hat{e}_0(S_1) \geq 10\varepsilon x$. For all t and for all substrings X of S_2 that are not rejected at Step 4(b) of* MakeDecision(x, ε, k)*, the subsets $\mathcal{L}_t$ and $\mathcal{R}_t(\tau)$ contain a pair of dense blocks with equal ranks with probability $> 1 - 1/9x^2$.*

Proof. Let $\mathcal{D}$ be the set of ranks m of blocks such that both the m-th block of S_1 and the m-th block of X are dense. From the assumption of the lemma we have $e_0(S_1) \geq 9\varepsilon x$ and since X is not rejected at Step 4(b), $e_1(X) > 9\varepsilon x$ as well. Therefore, the total number of all blocks in S_1 or X is at least $9\varepsilon x/11\varepsilon' x'$. The total number of non-dense blocks in both strings is at most $2x/x' = 22\varepsilon' x/11\varepsilon' x' < \varepsilon x/11\varepsilon' x'$. Therefore, $|\mathcal{D}| \geq 8\varepsilon x/11\varepsilon' x'$. On the other hand, the total number of blocks (and, in particular, dense blocks) in S_1 and X is at most $x/8\varepsilon' x'$. It follows that $|\mathcal{D}| \leq x/8\varepsilon' x'$.

Recall that both $\mathcal{L}_t$ and $\mathcal{R}_t(\tau)$ have size $\mathbf{B} \cdot (\varepsilon')^{-1}\sqrt{x/x'}\log(\varepsilon^{-1}x\log x)$. We view the sets as $\mathbf{B} \cdot \log(\varepsilon^{-1}x\log x)$ experiments during which we select two subsets of $\mathcal{D}$. Note that each dense block is selected with probability at least $((\varepsilon')^{-1}\sqrt{x/x'})/(x/8\varepsilon' x') \geq 8/\sqrt{x/x'}$. Therefore, the expectation of the size of the selected subsets of $\mathcal{D}$ is at least $8|\mathcal{D}|/\sqrt{x/x'}$. From the lower bound on $|\mathcal{D}|$ it follows that the latter is at least $2\sqrt{|\mathcal{D}|}$, and therefore the size of the selected subsets is at least $\sqrt{|\mathcal{D}|}$ with probability $> 3/4$ (this is a rough bound which is sufficient for our purposes). Recall that the Birthday paradox claims that any two subsets of $\mathcal{D}$ of size $\sqrt{|\mathcal{D}|}$ sampled uniformly without replacement contain equal elements with probability $> 1/2$. Therefore, in each experiment there is a pair of dense blocks with equal ranks with probability $> 1/4$. Since we repeat the experiments $\mathbf{B} \cdot \log(\varepsilon^{-1}x\log x)$ times, the probability that at least one of them is successful is at least $1 - 1/(9\mathbf{A} \cdot \varepsilon^{-1}x^2\log x)$ for a sufficiently large constant $\mathbf{B}$. By the union bound over all t the lemma holds with probability $> 1 - 1/9x^2$. □

Lemma 4.4. *If less than $e_0(S_1) - 30\varepsilon x$ excess opening parentheses of S_1 can be matched sequentially in S_2, then Algorithm 3 rejects S_1 and S_2 with probability $> 2/3$.*

Proof. We show the claim by induction on k. Suppose that MakeDecision$(\varepsilon, x, k-1)$ rejects with probability $> 2/3$ if run on two strings B_1, B_2 such that less than $e_0(B_1) - 30\varepsilon x$ excess opening parentheses of B_1 can be matched sequentially in B_2. We will show it implies that MakeDecision(x, ε, k) will reject S_1, S_2 with probability $> 2/3$. The lemma then follows, as the base case ($k = 0$) obviously holds.

Consider some substring X of S_2. We call a block of S_1 of rank m *bad* (with respect to X) if we cannot match more than $60(\varepsilon')^2 x'$ excess opening parentheses of it in the m-th block of X. Suppose that at most $\varepsilon/3$-fraction of blocks of S_1 are bad. We show that in this case we can match more than $e_0(S_1) - 30\varepsilon x$ excess opening parentheses of S_1 in X. Indeed, the total number of non-dense blocks of S_1 is at most x/x' and hence the total number of excess parentheses in non-dense blocks is at most $12\varepsilon' x = 12\varepsilon x/30 < \varepsilon x/2$. Consider the set of dense blocks of S_1. The leftmost and the rightmost blocks of S_1 can be dense but contain at most $24\varepsilon x$ excess opening parentheses. By our assumption, among the remaining blocks there is at most $\varepsilon/3$-fraction of bad blocks. Therefore, the number of excess parentheses in such blocks is at most $(x/8\varepsilon' x') \cdot (\varepsilon/3) \cdot 12\varepsilon' x' \leq \varepsilon x/2$. On the other hand, we can match all but $60(\varepsilon')^2 x'$ excess opening parentheses in each of the remaining blocks, or at most $60(\varepsilon')^2 x/8 = 5\varepsilon x/2$ parentheses in total. Therefore, the total number of unmatched excess parentheses is less than $30\varepsilon x$ as claimed.

It therefore suffices to show that if S_1 contains more than $\varepsilon/3$-fraction of bad blocks for each substring X of S_2, then it will be rejected with probability $> 2/3$. Equivalently, we can show that the probability to accept S_1 and S_2 is at most $1/3$. We can accept the strings either because we made an error while approximating the excess numbers (which can happen with probability $< 1/9$) or because S_1 and some substring X of S_2 are erroneously accepted. Since the length of S_2 is at most x, it has at most $x^2/2$ substrings. We will show that each of them is accepted with probability $< 2/9x^2$. The claim will follow by the union bound. From Lemma 4.3 it follows that we will find $\mathbf{A} \cdot \varepsilon^{-1}\log x$ pairs of dense blocks with equal ranks with probability at least $1 - 1/9x^2$. Since at

least $\varepsilon/3$-fraction of the blocks of S_1 is bad with respect to X, we can select the constant $\mathbf{A}$ so that at least one of the bad blocks is selected with probability $> 1 - 1/18x^2$. Finally, we can select the constant $\mathbf{C}$ so that the bad block is rejected by a recursive call to MakeDecision$(\varepsilon, x, k-1)$ with probability $> 1 - 1/18x$, which concludes the proof.

Note that we might need different values of $\mathbf{C}$ for this lemma and Lemma 4.2. We take the maximum of the values to ensure both lemmas. □

5 Lower bounds for Truestring equivalence and D_m-membership

In this section we prove the following lower bound for testing truestring equivalence.

Theorem 5.1. *Testing truestring equivalence requires at least $\Omega(n^{1/5})$ queries.*

Since the Truestring equivalence(n) problem can be reduced to the D_m-membership$(4n)$ problem by Lemma 2.5, we immediately obtain a similar lower bound for testing D_m-membership.

Corollary 5.2. *Testing D_m-membership requires at least $\Omega(n^{1/5})$ queries.*

For the lower bound construction let us introduce several definitions. Recall that for a string $w \in \{0,1,\diamond\}^*$, its *truestring* $T(w)$ is the subsequence resulting from deleting all "$\diamond$" characters. Given a string $u \in \{0,1\}^n$ and a set $U \in \binom{\{1,\ldots,2n\}}{n}$, we denote by $S(u,U)$ the unique string $w \in \{0,1,\diamond\}^{2n}$ for which $U = \{i : w[i] = \text{"}\diamond\text{"}\}$ and $u = T(w)$.

Definition 5.3 (Positive and negative distributions)**.** *We let $u \in \{0,1\}^n$ be chosen uniformly at random (every $u[i]$ independently), and let $u' \in \{0,1\}^n$ be the string resulting from replacing $u[i]$ with another uniformly and independently random member of $\{0,1\}$ for every $2n/5 < i < 3n/5$. Let $U \in \binom{\{1,\ldots,2n\}}{n}$ be a random set defined by choosing independently and uniformly whether $i \in U$ and $2n+1-i \notin U$, or $i \notin U$ and $2n+1-i \in U$, for every $1 \le i \le n$. Let $U' \in \binom{\{1,\ldots,2n\}}{n}$ be a second set chosen independently using the same distribution as that used for the choice of U. For the distribution $\mathcal{D}_P$, we set $w = S(u,U)$ and $w' = S(u,U')$. For the distribution $\mathcal{D}_N$, we set $w = S(u,U)$ and $w' = S(u',U')$.*

Lemma 5.4. *$\mathcal{D}_P$ is supported over string pairs that are truestring equivalent, while $\mathcal{D}_N$ with probability $1 - o(1)$ produces a pair that is $1/200$-far from truestring equivalence.*

Proof. The first part of the statement is immediate. The second part follows from the fact that two strings of length $n/5$ drawn uniformly and independently at random will have an edit distance between them of at least $n/100$. For showing this consider all $\binom{n/5}{n/100}^2 = o(2^{16n/100})$ possible ways of deleting $n/100$ characters from the first string and $n/100$ characters from the second string. For every such possibility the probability for the remaining strings to match is $\Theta(2^{-19n/100})$. A union bound concludes the argument. □

In the rest of the section we prove the next lemma, which by Yao's argument implies (together with Lemma 5.4) Theorem 5.1.

Lemma 5.5. *Any deterministic algorithm making $o(n^{1/5})$ queries will have acceptance probabilities for $\mathcal{D}_P$ and $\mathcal{D}_N$ that differ by $o(1)$.*

Toward the proof of this lemma, we use the following definitions in the analysis.

Definition 5.6 (True index)**.** *Given a string w and $1 \leq i \leq 2n$ for which $w[i] \neq \diamond$, the* true index *$t_w(i)$ is defined as $|\{j \leq i : w[j] \neq \diamond\}|$. In other words, it is the index j such that $w[i]$ determines $(T(w))[j]$.*

The next definition is adapted for our particular distributions $\mathcal{D}_P$ and $\mathcal{D}_N$, where for every i and i' exactly one of $w[i]$ and $w[2n+1-i]$ is $\diamond$ and exactly one of $w'[i']$ and $w'[2n+1-i']$ is $\diamond$.

Definition 5.7 (Matching indexes)**.** *Given the strings w and w', and the indexes $1 \leq i \leq 2n$ and $1 \leq i' \leq 2n$, we call (i, i')* matching *with respect to (w, w') if either $w[i], w'[i'] \neq \diamond$ and $t_w(i) = t_{w'}(i')$, or $w[2n+1-i], w'[2n+1-i'] \neq \diamond$ and $t_w(2n+1-i) = t_{w'}(2n+1-i')$ (we omit below the pair notation, e.g. "(i, i')" and "(w, w')", whenever it is clear which index among i and i' refers to w and which index refers to w').*

Ironically, the analysis of a deterministic adaptive algorithm becomes much easier if we allow the input query oracle to "leak" additional information when answering a query. We define what we mean below. The following definition is valid for any finite domain, but for our purposes we define a domain $D := \{1, \ldots, 2n\} \times \{1, 2\}$, referring to two length-$2n$ strings.

Definition 5.8 (Super-oracle)**.** *A* super-oracle *for an input W over the domain D is an algorithm, intended to be called whenever a query to W is made, satisfying the following features.*

- *The algorithm holds W and possibly other variables (i.e. it is not stateless), which are given initial values before the first query.*
- *Whenever a query $i \in D$ is made, the algorithm provides a set of indexes I and all values of $W|_I$. It is mandated that $i \in I$, and assumed (without loss of generality) that I also contains all indexes provided in the previous query (except when responding to the very first query).*

It is immediate that a testing algorithm (working with a "regular" oracle) can be converted to a testing algorithm working with a super-oracle $\mathcal{Q}$, by simply ignoring the additional information, which leads to the following observation.

Observation 5.9. *To prove Lemma 5.5, it is enough to construct a super-oracle $\mathcal{Q}$, and prove that the distributions over the transcript of any algorithm $\mathcal{A}$ with $q = o(n^{1/5})$ queries running against $\mathcal{Q}$, when the input is drawn by either $\mathcal{D}_P$ or $\mathcal{D}_N$, are $o(1)$-close to each other.*

Another assumption that is without loss of generality, and that we make from now on, is that the algorithm $\mathcal{A}$ never makes a query which has already been revealed in response to a previous query (the algorithm can just read internally the value that has already been given), and in particular we assume that the algorithm terminates immediately (and gives the correct answer) when $\mathcal{Q}$ reveals the entirety of the input W. We denote this "total revelation" event by "$\perp$".

The following observation about deterministic algorithms interacting with queries is well-known for direct queries, and also holds (with the same proof) for algorithms interacting with a specific super-oracle.

Observation 5.10. *For a deterministic algorithm $\mathcal{A}$ working against any super-oracle $\mathcal{Q}$ with any input W, under the (without loss of generality) assumptions that the set revealed by $\mathcal{Q}$ in response to a query always contains the previously revealed set and that $\mathcal{A}$ never queries a previously revealed location, the transcript of the algorithm is fully determined by the final revealed set I and the values $W|_I$ of the input over it.*

Additionally, the last revealed set I and the values $W|_I$ of the input over it (as provided to $\mathcal{A}$ by $\mathcal{Q}$) completely determine its state and its transcript so far (including the number of rounds already executed).

The above observation however is not as useful as it seems at first, since (unlike the setting where an algorithm queries the input without a super-oracle) the probability of an algorithm to reach a set characterized by I and $W|_I$ where W is drawn from some distribution $\mathcal{D}$ is not necessarily the probability of the input to have the prescribed values over I. The reason is that I itself might depend on values of the input outside this set. However, we will construct a super-oracle that satisfies an additional feature that helps.

Definition 5.11. *A super-oracle is called* oblivious *if at any point, either the entirety of the input is revealed (the "$\perp$" event), or the revealed set I depends only on the history of the algorithm's query locations and the input values on the revealed locations (the conditions for triggering the "$\perp$" event can still depend on the entirety of the input).*

This allows us to restore the convenient analysis of algorithms as decision trees, with a caveat concerning the $\perp$ event which we have to analyze separately.

Observation 5.12. *A deterministic algorithm $\mathcal{A}$ working with an oblivious super-oracle $\mathcal{Q}$ corresponds to a decision tree, where every leaf is labeled with an output ("accept" or "reject"), every inner node is labeled by the next revealed set I (since $\mathcal{Q}$ is oblivious, the set is determined only by the history on the path leading to it unless $\perp$ has been triggered), and every edge from a node to a child corresponds to a possible set of values $W|_I$.*

Accordingly, when the input is drawn from a distribution $\mathcal{D}$, the probability of reaching a certain leaf whose parent contains the revealed set I (containing the sets of its ancestors) and whose incoming edge corresponds to some value of $W|_I$, is the probability of the intersection of the following two events: The event of an input drawn from $\mathcal{D}$ to have the corresponding values over I, and the negation of the "$\perp$" (total reveal) event.

The exact calculation in the above observation is still complex, but we will only need to have upper bounds as per the following definition.

Definition 5.13. *Given a set S and an additional symbol "$\perp$", we say that a distribution $\mathcal{D}$ over $S \cup \{\perp\}$* underlies *a distribution $\mathcal{C}$ over $S \cup \{\perp\}$ if for every $a \in S$ (not including "$\perp$") we have $\mathcal{D}(a) \leq \mathcal{C}(a)$.*

The following is immediate.

Observation 5.14. *If $\mathcal{D}$ underlies $\mathcal{C}$, both being distributions over $S \cup \{\perp\}$, then the variation distance between them is at most $\mathcal{D}(\perp)$.*

Proof. The distance between the two distributions is

$$\frac{1}{2} \sum_{a \in S \cup \{\perp\}} |\mathcal{D}(a) - \mathcal{C}(a)| = \sum_{\{a \in S \cup \{\perp\} : \mathcal{D}(a) > \mathcal{C}(a)\}} (\mathcal{D}(a) - \mathcal{C}(a))$$

In our case, the right hand is a sum over a set that may only contain $\perp$ itself, so the difference is bounded by $\mathcal{D}(\perp) - \mathcal{C}(\perp) \leq \mathcal{D}(\perp)$. □

In our case, we analyse the following distributions.

Definition 5.15. *Given an algorithm $\mathcal{A}$, a super oracle $\mathcal{Q}$ and a distribution $\mathcal{D}$ over inputs, we denote by $\mathcal{A}^{\mathcal{Q}}_{\mathcal{D}}$ the following distribution over $L \cup \{\perp\}$, where L is the set of leaves of the decision tree corresponding to $\mathcal{A}$ and $\mathcal{Q}$ by Observation 5.12:*

- *For $\ell \in L$, $\mathcal{A}^{\mathcal{Q}}_{\mathcal{D}}(\ell)$ equals the probability of reaching ℓ (without triggering the $\perp$ event),*

- $\mathcal{A}^{\mathcal{Q}}_{\mathcal{D}}(\perp)$ *equals the probability of the* $\perp$ *event being triggered (which always implies a correct output by the algorithm).*

For the distributions $\mathcal{D}_P$ *and* $\mathcal{D}_N$ *defined above, we use the shorthand* $\mathcal{A}^{\mathcal{Q}}_P$ *and* $\mathcal{A}^{\mathcal{Q}}_N$ *respectively for* $\mathcal{A}^{\mathcal{Q}}_{\mathcal{D}_P}$ *and* $\mathcal{A}^{\mathcal{Q}}_{\mathcal{D}_N}$.

We have to be careful about the closeness guarantees if we want to use Observation 5.9.

Observation 5.16. *To prove Lemma 5.5, it is enough to construct a super-oracle* $\mathcal{Q}$*, and prove that* $\mathcal{A}^{\mathcal{Q}}_P(\perp) = o(1)$*, that* $\mathcal{A}^{\mathcal{Q}}_N(\perp) = o(1)$*, and that* $\mathcal{A}^{\mathcal{Q}}_P$ *and* $\mathcal{A}^{\mathcal{Q}}_N$ *are* $o(1)$*-close to each other.*

Proof. We use Observation 5.9, but note that the $\perp$ event does not imply identical outputs of $\mathcal{A}$ under $\mathcal{D}_P$ and $\mathcal{D}_N$. In fact it implies quite the opposite (the $\perp$ event causes $\mathcal{A}$ to provide the correct output, which is different for the two distributions). The distance between the algorithmic behaviors under the two distributions is bounded by $d(\mathcal{A}^{\mathcal{Q}}_P, \mathcal{A}^{\mathcal{Q}}_N) + \frac{1}{2}\left(\mathcal{A}^{\mathcal{Q}}_P(\perp) + \mathcal{A}^{\mathcal{Q}}_N(\perp)\right)$, which by the assertions of the observation is $o(1)$. □

We now define a specific super-oracle $\mathcal{Q}$ to be used with an input drawn by either $\mathcal{D}_P$ or $\mathcal{D}_N$ (defined above), and denote by $\mathcal{A}^{\mathcal{Q}}_P$ and $\mathcal{A}^{\mathcal{Q}}_N$ the respective distributions over transcripts when a deterministic algorithm $\mathcal{A}$ is run against $\mathcal{Q}$ with the input drawn by $\mathcal{D}_P$ and $\mathcal{D}_N$. We also define a super-oracle $\mathcal{Q}^-$ for the sake of some interim lemmas.

Definition 5.17. *The super-oracle* $\mathcal{Q}$ *is defined as per Algorithm 4. The super-oracle* $\mathcal{Q}^-$ *is defined identically with the exception that Step 4 is not run (meaning that the oracle never performs a "total reveal" of the input).*

Algorithm 4 Super-oracle $\mathcal{Q}$ for truestring equivalence

Query provider for: Input strings $w, w' : \{1, \ldots, 2n\} \to \{0, 1\}$
Persistent variables: $i_m \in \{0, \ldots, n\}$, initialized to 0, and $Q, Q' \subseteq \{1, \ldots, n\}$, both initialized to $\emptyset$
Activation parameter: A query j from w or a query j' from w'

1. If a query $j \in \{1, \ldots, 2n\}$ is requested from w, add $\min\{j, 2n + 1 - j\}$ to Q
2. If a query $j' \in \{1, \ldots, 2n\}$ is requested from w', add $\min\{j', 2n + 1 - j'\}$ to Q'
3. While there exists $j \in Q \cup Q'$ with $j \leq i_m + 100n^{4/5}$, set i_m to $\max\{j, i_m\}$, set Q to $Q \setminus \{1, \ldots, j\}$, and set Q' to $Q' \setminus \{1, \ldots, j\}$
4. If there exist $j \in Q$ and $j' \in Q'$ that are matching with respect to w and w', reveal the entirety of w and w' and terminate
5. Reveal all values in $\{1, \ldots, i_m, 2n + 1 - i_m, \ldots, 2n\} \cup \bigcup \left\{\{j, 2n + 1 - j\} : j \in Q\right\}$ from w and all values in $\{1, \ldots, i_m, 2n + 1 - i_m, \ldots, 2n\} \cup \bigcup \left\{\{j', 2n + 1 - j'\} : j' \in Q'\right\}$ from w'

The following observations are immediate.

Observation 5.18. *The super-oracle* $\mathcal{Q}$ *of Algorithm 4 is oblivious (and so is* $\mathcal{Q}^-$*). In fact, outside the* $\perp$ *event, the revealed set only depends on the queried locations and not at all on the input (the* $\perp$ *event itself as per Step 4 of the algorithm can depend on the entire input).*

Observation 5.19. *The distribution* $\mathcal{A}^{\mathcal{Q}}_N$ *underlies the distribution* $\mathcal{A}^{\mathcal{Q}^-}_N$.

Proof. We note that the tree corresponding to $\mathcal{A}$ running with $\mathcal{Q}^-$ can be used as the tree for $\mathcal{A}$ running with $\mathcal{Q}$, since the only difference is that some inputs can trigger $\perp$ events in some tree nodes with $\mathcal{Q}$ (but not with $\mathcal{Q}^-$). The observation then follows from a direct application of Observation 5.12. □

We next note that querying an index i from w will always reveal both $w[i]$ and $w[2n+1-i]$ (and similarly for querying an index from w'), and henceforth assume that all queries are made in the range $\{1,\ldots,n\}$. The following lemma bounds the increase of i_m during q interaction rounds.

Lemma 5.20. *After q interaction rounds of a testing algorithm $\mathcal{A}$ against the super-oracle $\mathcal{Q}$ we have $i_m \leq q \cdot 100n^{4/5}$.*

Proof. Note that every round can add at most one index to $Q \cup Q'$. Any increase in i_m involves the removal of at least one index from $Q \cup Q'$ (which could happen at a later round than the round where that index was inserted), and (by the condition of Step 3) each such removal increases i_m by at most $100n^{4/5}$. The lemma follows. □

For given $U, U' \subset \{1,\ldots,2n\}$ which have a positive probability of being drawn as per the definition of $\mathcal{D}_P$ and $\mathcal{D}_N$, we define $\mathcal{A}^{\mathcal{Q}}_{P:U,U'}$ and $\mathcal{A}^{\mathcal{Q}}_{N:U,U'}$ as the respective distributions derived from $\mathcal{A}^{\mathcal{Q}}_P$ and $\mathcal{A}^{\mathcal{Q}}_N$ when we condition the drawing of the input on the specific U and U'. We define $\mathcal{A}^{\mathcal{Q}^-}_{P:U,U'}$ and $\mathcal{A}^{\mathcal{Q}^-}_{N:U,U'}$ analogously. The following lemma is a crucial companion to Observation 5.19.

Lemma 5.21. *For any algorithm making $q = o(n^{1/5})$ queries and n large enough, the distribution $\mathcal{A}^{\mathcal{Q}}_P$ also underlies the distribution $\mathcal{A}^{\mathcal{Q}^-}_N$.*

Proof. For n large enough we have $q \cdot 100n^{4/5} < 2n/5$, meaning by Lemma 5.20 that i_m (and $2n+1-i_m$) never reach a range in w or w' where the respective u and u' (as drawn in the definition of $\mathcal{D}_P$ and $\mathcal{D}_N$) differ. Additionally, $\mathcal{Q}^-$ never reveals the entire input. As with the proof of Observation 5.19, we note that the tree corresponding to $\mathcal{A}$ running with $\mathcal{Q}^-$ can be used as the tree for $\mathcal{A}$ running with $\mathcal{Q}$. Let us now consider a leaf of this tree. It corresponds to the revealed set of its parent, calculated in Step 5 of Algorithm 4 using i_m, Q and Q', and the corresponding values of the input, namely $w|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\} \cup \bigcup\{\{j,2n+1-j\}:j\in Q\}}$ and $w'|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\} \cup \bigcup\{\{j',2n+1-j'\}:j'\in Q'\}}$.

Recall that an input drawn by $\mathcal{D}_P$ or $\mathcal{D}_N$ is a function of the sets $U, U' \subseteq \{1,\ldots,2n\}$ and the strings $u, u' \in \{0,1\}^n$. Recall also that U and U' are drawn in the same way and play the same role in the two distributions, and that additionally the total reveal behaviour of $\mathcal{Q}$ depends only on U and U' (which determine the true indexes in w and w') and the query history, and does not depend at all on u or u'. Hence, to show for some leaf a that $\mathcal{A}^{\mathcal{Q}}_P(a) \leq \mathcal{A}^{\mathcal{Q}^-}_N(a)$, it is enough to consider any U and U' that are consistent with the revealed parts of w and w' and do not cause $\mathcal{Q}$ to reveal the entire input (by Algorithm 4, Step 4) on the respective root-to-leaf path, and show that the corresponding conditional probabilities satisfy $\mathcal{A}^{\mathcal{Q}}_{P:U,U'}(a) \leq \mathcal{A}^{\mathcal{Q}^-}_{N:U,U'}(a)$.

In fact, we show that the above two conditional probabilities are identical, by showing that the respective conditional distributions over $w|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\} \cup \bigcup\{\{j,2n+1-j\}:j\in Q\}}$ and $w'|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\} \cup \bigcup\{\{j',2n+1-j'\}:j'\in Q'\}}$ are identical (recall that in the decision tree, when it is guaranteed that the $\perp$ event cannot be triggered along the path, reaching the leaf a is equivalent to having the respective values over the revealed sets).

We now analyze the distribution over the revealed values of w and w' when U and U' are fixed as above, and u and u' are drawn from $\{0,1\}^n$ according to the definition of $\mathcal{D}_P$ and $\mathcal{D}_N$ (which as we recall, are independent of U and U'). Note that u is drawn (as per the definition of both $\mathcal{D}_P$ and $\mathcal{D}_N$) uniformly from $\{0,1\}^n$, which (since $i_m \leq 2n/5$) together with U and U' determines all values of $w|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\}}$ and $w'|_{\{1,\ldots,i_m,2n+1-i_m,\ldots,2n\}}$ (recall that u is identical to u' on the relevant parts).

Next, for every $j \in Q$ which matches some $1 \leq i' \leq i_m$, again the values of w at j and $2n+1-j$ are completely determined by the already drawn sets (U determines which of them is "$\diamond$" and which of them takes the non-$\diamond$ value among $w'[i']$ and $w'[2n+1-i']$, the latter value being equal to either

$u'[t_{w'}(i')] = u[t_{w'}(i')]$ or $u'[t_{w'}(2n+1-i')] = u[t_{w'}(2n+1-i')]$, which were already treated above). An analogous consideration holds for any $j' \in Q'$ which matches some $1 \leq i \leq i_m$. Denote by $\tilde{Q}$ and $\tilde{Q}'$ respectively the remaining indexes in Q and Q', namely those not matching any index between 1 and i_m.

Considering the values of w and w' on indices in $\tilde{Q}$ and $\tilde{Q}'$ (which are the only values of w and w' not treated above), note that by $\perp$ not occurring, none of them matches any other index in the revealed parts of w and w'. This means that for any $j \in \tilde{Q}$, the value among $w[j]$ and $w[2n+1-j]$ which is not "$\diamond$" (as determined by U) is distributed uniformly over $\{0,1\}$ and independently of all other values that need to be revealed, and same goes for any $j' \in \tilde{Q}'$ not covered above with respect to the value among $w'[j']$ and $w'[2n+1-j']$ which is not "$\diamond$" (as determined by U'). This again holds over both $\mathcal{D}_P$ and $\mathcal{D}_N$.

We have shown here that for every possible revealed set outside the total-reveal "$\perp$" event, the probability for any revealed input is the same for $\mathcal{D}_P$ and $\mathcal{D}_N$, which as discussed above in particular means that $\mathcal{A}^{\mathcal{Q}}_{P:U,U'}(a) \leq \mathcal{A}^{\mathcal{Q}^-}_{N:U,U'}(a)$, and hence (by considering all U, U' that allow $\mathcal{A}$ to reach a over $\mathcal{Q}$, which form a subset of the corresponding set over $\mathcal{Q}^-$) we have $\mathcal{A}^{\mathcal{Q}}_P(a) \leq \mathcal{A}^{\mathcal{Q}^-}_N(a)$. □

It remains to show that the total reveal event $\perp$ happens with probability $o(1)$ as n grows large. To give a good bound on two specific indexes from Q and Q' matching each other, we use the following technical lemma that follows from Stirling's formula.

Lemma 5.22. *If $X_1, \ldots, X_m$ are independent random variables with values chosen uniformly from $\{0,1\}$, then for m large enough and all j we have $\Pr[\sum_{i=1}^m X_i = j] \leq 1/\sqrt{m}$.*

In our super-oracle (Algorithm 4) we make sure (by Step 3) that the $\perp$ event can only be triggered by an index that is not "too close" to the "revealed range" as it relates to i_m (see Algorithm 4, Step 5). Such indexes are subject to the following bound.

Lemma 5.23. *For n large enough, given any $1 \leq i \leq n$, $i + 100n^{4/5} < j \leq n$, and $Q' \subset \{1, \ldots, n\}$ with $\min Q' > i + 100n^{4/5}$, the probability (under both $\mathcal{D}_P$ and $\mathcal{D}_N$) that j matches any $j' \in Q'$ with respect to w and w', is bounded by $|Q'|/5n^{2/5}$, also when conditioned on the entirety of w and the values $w'|_{\{1,\ldots,i,2n+1-i,\ldots,2n\} \cup \bigcup\{\{j',2n+1-j'\}:j' \in Q'\}}$. The analogous statement holds for $i + 100n^{4/5} < j' \leq n$ and $Q \subset \{1, \ldots, n\}$ with $\min Q > i + 100n^{4/5}$ when conditioning on the entirety of w' and the values $w|_{\{1,\ldots,i,2n+1-i,\ldots,2n\} \cup \bigcup\{\{j,2n+1-j\}:j \in Q\}}$.*

Proof. We prove the lemma for the first statement, since the proof for the second statement is indeed analogous (swapping w and w'). Fixing some $j' \in Q'$, since the event of j matching j' depends only on the identity of U and U' (which have the same distribution under both $\mathcal{D}_P$ and $\mathcal{D}_N$), the following analysis holds for both distributions. Recall that exactly one of $w[j]$ and $w[2n+1-j]$ is "$\diamond$", and the same goes for exactly one of $w'[j']$ and $w'[2n+1-j']$. We next prove that the probability that j matches j' is at most $1/5n^{2/5}$. To do so, we prove that the bound holds even if we additionally condition on any value of $w'|_{\{\min Q', \ldots, j'\}}$. The bound is proved by considering each of the following four cases.

Consider first the case where $w[j], w'[j'] \neq \diamond$. Let X_k be the indicator variable for "$w'[k+i] \neq \diamond$" for $1 \leq k \leq m$, where $m = \min Q' - i - 1 \geq 100n^{4/5} - 1 > 25n^{4/5}$. The locations j and j' match if and only if

$$\sum_{k=1}^{m} X_k = |\{1 \leq l < j : w[l] \neq \diamond\}| - |\{1 \leq l \leq i : w'[l] \neq \diamond\}| - |\{\min Q' \leq l < j' : w'[l] \neq \diamond\}|$$

By Lemma 5.22, the probability of this event is at most $1/\sqrt{25n^{4/5}} = 1/5n^{2/5}$.

In the case where $w[j] = w'[j'] = \diamond$ (where j and j' can match due to the true indexes at $2n+1-j$ and $2n+1-j'$), we again use Lemma 5.22, where each X_k is the indicator variable for "$w'[2n+1-k-i] \neq \diamond$" for $1 \leq k \leq m$, where $m = \min Q' - i - 1 \geq 100n^{4/5} - 1 > 25n^{4/5}$. We then have that j, j' match if and only if

$$\sum_{k=1}^{m} X_k \;=\; |\{2n+1-j < l \leq 2n : w[l] \neq \diamond\}| \;-\; \left|\left\{2n+1-i \leq l \leq 2n : w'[l] \neq \diamond\right\}\right| \\ -\; \left|\left\{2n+1-j' < l \leq 2n+1-\min Q' : w'[l] \neq \diamond\right\}\right|$$

By Lemma 5.22, the probability of this event is at most $1/\sqrt{25n^{4/5}} = 1/5n^{2/5}$.

In the final two cases, where exactly one of $w[j]$ and $w'[j']$ is "$\diamond$", no matching can occur. Thus the probability for a match between j and j' is bounded by $1/5n^{2/5}$. To finalize, we consider the union bound of this event for every $j' \in Q'$, to arrive at the $|Q'|/5n^{2/5}$ bound for the probability that j matches any $j' \in Q'$. □

It is now easy to prove that the probability for $\perp$ under both $\mathcal{D}_P$ and $\mathcal{D}_N$ is small.

Lemma 5.24. *For any algorithm $\mathcal{A}$ making $q = o(n^{1/5})$ queries and the super-oracle $\mathcal{Q}$, we have $\mathcal{A}_P^{\mathcal{Q}}(\perp) = o(1)$ and $\mathcal{A}_N^{\mathcal{Q}}(\perp) = o(1)$.*

Proof. This result follows from a simple union bound using the per-round bound of Lemma 5.23, noting that $q^2/5n^{2/5} = o(1)$. □

Proof of Lemma 5.5 (implying Theorem 5.1 and Corollary 5.2). Given any algorithm $\mathcal{A}$ making $q = o(n^{1/5})$ queries, by Lemma 5.24 the probability of $\perp$ (the total reveal event) under both distributions $\mathcal{A}_P^{\mathcal{Q}}$ and $\mathcal{A}_N^{\mathcal{Q}}$ is $o(1)$. This implies by Observation 5.19 and Observation 5.14 that $d(\mathcal{A}_N^{\mathcal{Q}}, \mathcal{A}_N^{\mathcal{Q}^-}) = o(1)$, and by Lemma 5.21 and Observation 5.14 that $d(\mathcal{A}_P^{\mathcal{Q}}, \mathcal{A}_N^{\mathcal{Q}^-}) = o(1)$. Thus $d(\mathcal{A}_P^{\mathcal{Q}}, \mathcal{A}_N^{\mathcal{Q}}) = o(1)$.

Putting the above together we have all the components to deduce by Observation 5.16 the validity of Lemma 5.5. □

6 Acknowledgements

We thank Sofya Raskhodnikova, Diptaksho Palit, and Timothy Jackman for pointing out a flaw in our lower bound proof, which enabled us to correct it.

References

[1] N. Alon, M. Krivelevich, I. Newman, and M. Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, May 2009.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, May 1998.

[4] J.-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, pages 111–174. Springer Berlin Heidelberg, 1997.

[5] T. Batu, L. Fortnow, R. Rubinfeld, W.D. Smith, and P. White. Testing closeness of discrete distributions. *J. ACM*, 60(1):4:1–4:25, February 2013.

[6] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.

[7] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993.

[8] K. Bringmann, F. Grandoni, B. Saha, and V.V. Williams. Truly sub-cubic algorithms for language edit distance and RNA-folding via fast bounded-difference min-plus product. In *Proc. of the IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS 2016)*, pages 375–384, 2016.

[9] M. Chu, S. Kannan, and A. McGregor. Checking and spot-checking the correctness of priority queues, 2007.

[10] A. Czumaj, C. Sohler, and M. Ziegler. Property testing in computational geometry. In *Proc. of the 8th Annual European Symposium on Algorithms (ESA 2000)*, pages 155–166, 2000.

[11] P.W. Dymond and W.L. Ruzzo. Parallel RAMs with owned global memory and deterministic context-free language recognition. *J. ACM*, 47(1):16–45, January 2000.

[12] F. Ergün, S. Kannan, S. R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *Journal of Computer and System Sciences*, 60(3):717 – 751, 2000.

[13] E. Fischer, F. Magniez, and M. de Rougemont. Approximate satisfiability and equivalence. *SIAM Journal on Computing*, 39(6):2251–2281, 2010.

[14] N. François, F. Magniez, M. de Rougemont, and O. Serre. Streaming property testing of visibly pushdown languages. In *Proc. of the 24th Annual European Symposium on Algorithms (ESA 2016)*, pages 43:1–43:17, 2016.

[15] O. Goldreich, S. Goldwasser, E. Lehman, D. Ron, and A. Samorodnitsky. Testing monotonicity. *Combinatorica*, 20(3):301–337, Mar 2000.

[16] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, July 1998.

[17] O. Goldreich and D. Ron. Property testing in bounded degree graphs. *Algorithmica*, 32(2):302–343, Feb 2002.

[18] F. Magniez, C. Mathieu, and A. Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014.

[19] A. Montanaro and R. de Wolf. A survey of quantum property testing. *Graduate Surveys*, 7:1–81, 2016.

[20] M. Parnas, D. Ron, and R. Rubinfeld. Testing membership in parenthesis languages. *Random Structures & Algorithms*, 22(1):98–138, 2003.

[21] R. Rubinfeld. On the robustness of functional equations. *SIAM Journal on Computing*, 28(6):1972–1997, 1999.

[22] R. Rubinfeld and M. Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.

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

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

[25] B. von Braunmühl, S. Cook, K. Mehlhorn, and R. Verbeek. The recognition of deterministic CFLs in small time and space. *Information and Control*, 56(1):34 – 51, 1983.