Quantum Information Processing (2023) 22:194
https://doi.org/10.1007/s11128-023-03910-9

Quantum bounds for 2D-grid and Dyck language

Andris Ambainis[1] · Kaspars Balodis[1] · Jānis Iraids[1] · Kamil Khadiev[2] · Vladislavs Kļevickis[1] · Krišjānis Prūsis[1] · Yixin Shen[3] · Juris Smotrovs[1] · Jevgēnijs Vihrovs[1]

Received: 4 November 2021 / Accepted: 16 March 2023

Abstract
We study the quantum query complexity of two problems. First, we consider the problem of determining whether a sequence of parentheses is a properly balanced one (*a Dyck word*), with a depth of at most k. We call this the $\textsc{Dyck}_{k,n}$ problem. We prove a lower bound of $\Omega(c^k\sqrt{n})$, showing that the complexity of this problem increases exponentially in k. Here n is the length of the word. When k is a constant, this is interesting as a representative example of star-free languages for which a surprising $\tilde{O}(\sqrt{n})$ query quantum algorithm was recently constructed by Aaronson et al. (Electron Colloquium Comput Complex (ECCC) 26:61, 2018). Their proof does not give rise to a general algorithm. When k is not a constant, $\textsc{Dyck}_{k,n}$ is not context-free. We give an algorithm with $O\left(\sqrt{n}(\log n)^{0.5k}\right)$ quantum queries for $\textsc{Dyck}_{k,n}$ for all k. This is better than the trivial upper bound n for $k = o\left(\frac{\log(n)}{\log\log n}\right)$. Second, we consider connectivity problems on grid graphs in 2 dimensions, if some of the edges of the grid may be missing. By embedding the "balanced parentheses" problem into the grid, we show a lower bound of $\Omega(n^{1.5-\epsilon})$ for the directed 2D grid and $\Omega(n^{2-\epsilon})$ for the undirected 2D grid. We present two algorithms for particular cases of the problem. The directed problem is interesting as a black-box model for a class of classical dynamic programming strategies including the one that is usually used for the well-known edit distance problem. We also show a generalization of this result to more than 2 dimensions

Keywords Quantum query complexity · Quantum algorithms · Dyck language · Grid path

The paper is an extended version of the conference paper [1] that was presented at MFCS2020.

✉ Kamil Khadiev
kamilhadi@gmail.com

✉ Yixin Shen
yixin.shen@rhul.ac.uk

Extended author information available on the last page of the article

Published online: 29 April 2023

1 Introduction

Quantum computers offer computational advantage in solving several classes of problems. The most famous example is the factorization of large integers [2]. The exponential advantage of simulating quantum physical systems is also a well-known example [3, 4]. This naturally leads to the question of understanding how large the advantage of quantum computers can be. Unfortunately, it is often very difficult to prove unconditional separations between classical and quantum complexity classes. For this reason, people often study the power of quantum computing in the query model. In this model, we want to compute a function $f(x_1, \ldots, x_n)$ of an input $(x_1, \ldots, x_n)$. The input x_i's are accessed via queries to a black box, that given i, outputs x_i. The complexity is measured by the number of queries that an algorithm makes. The query model is very interesting in the quantum case because it captures most of the known quantum algorithms. We refer to [5] for a nice survey on the topic.

In this paper, we study the quantum query complexity of two problems.

1.1 Quantum complexity of regular languages

Consider the problem of recognizing whether an n-bit string belongs to a given regular language. This models a variety of computational tasks that can be described by regular languages. In theoretical computer science and formal language theory, a regular language is a formal language that can be defined by a regular expression or can be defined as a language recognized by a finite automaton. See, for example, [6] for more details. In the quantum case, the most commonly used model for studying the complexity of various problems is the query model [5]. For this setting, Aaronson et al. [7] recently showed that any regular language L has one of three possible quantum query complexities on inputs of length n: $\Theta(1)$ if the language can be decided by looking at $O(1)$ first or last symbols of the word; $\tilde{\Theta}(\sqrt{n})$ if the best way to decide L is Grover's search (for example, for the language consisting of all words containing at least one letter a); $\Theta(n)$ for languages in which we can embed counting modulo some number p which has quantum query complexity $\Theta(n)$.

As shown in [7], a regular language being of complexity $\tilde{O}(\sqrt{n})$ (which includes the first two cases above) is equivalent to it being star-free. Star-free languages are defined as languages that have regular expressions not containing the Kleene star (if it is allowed to use the complement operation). Informally, Kleene star means "zero or more repetitions" in regular expressions, see [6] for more details. Star-free languages are one of the most commonly studied subclasses of regular languages, and there are many equivalent characterizations of them. One class of the star-free languages mentioned in [7] is the Dyck languages (with one type of parenthesis) with constant height k. Dyck language with height k consists of words with a balanced number of parentheses such that in no prefix the number of opening parentheses exceeds the number of closing parentheses by more than k; we denote the problem of determining whether an input of length n belongs to this language by $\textsc{Dyck}_{k,n}$. In the case of unbounded height $k = \frac{n}{2}$, the language is a fundamental example of a context-free language that is not regular. When more types of parenthesis are allowed, the famous Chomsky–Schützenberger representation theorem shows that any context-free language is the homomorphic image of the intersection of a Dyck language and a regular language [8]. Quantum algorithm for Dyck language with more types of parenthesis was investigated in [9].

1.2 Our results

We show that an exponential dependence of the complexity on k is unavoidable. Namely, for the balanced parentheses language, we have

- There exists $c > 1$ such that, for all $k \leq \log n$, the quantum query complexity is $\Omega(c^k \sqrt{n})$;
- If $k = c \log n$ for an appropriate constant c, the quantum query complexity is $\Omega(n^{1-\epsilon})$.

Thus, the exponential dependence on k is unavoidable and distinguishing sequences of balanced parentheses of length n and depth $\log n$ is almost as hard as distinguishing sequences of length n and arbitrary depth.

Similar lower bounds have recently been independently proven by Buhrman et al. [10].

Additionally, we give an explicit algorithm (see Theorem 3) for the decision problem $\textsc{Dyck}_{k,n}$ with $O\left(\sqrt{n}(\log n)^{0.5k}\right)$ quantum queries. The algorithm also works when k is not a constant and is better than the trivial upper bound of n when $k = o\left(\frac{\log(n)}{\log\log n}\right)$.

Note that classical (deterministic or randomized) query complexity of $\textsc{Dyck}_{k,n}$ is $\Omega(n)$ even if $k \geq 2$. In the case of $k = 2$, the problem is equivalent to an unstructured search among n elements whose lower bound in classical case is linear [11]. So, the presented quantum algorithm is better than classical counterparts in the case of $k = o\left(\frac{\log(n)}{\log\log n}\right)$.

1.3 Finding paths on a grid

The second problem that we consider is graph connectivity on subgraphs of the 2D grid. Consider a 2D grid with vertices (i, j), $i \in \{0, 1, \ldots, n\}$, $j \in \{0, 1, \ldots, k\}$ and edges from (i, j) to $(i+1, j)$ and $(i, j+1)$. The grid can be either directed (with edges in the directions of increasing coordinates) or undirected. We are given an unknown subgraph G of the 2D grid and we can perform queries to variables x_u (where u is an edge of the grid) defined by $x_u = 1$ if u belongs to G and 0 otherwise. The task is to determine whether G contains a path from $(0, 0)$ to (n, k).

Our interest in this problem is driven by the edit distance problem. In the edit distance problem, we are given two strings x and y and have to determine the smallest number of operations (replacing one symbol with another, removing a symbol or inserting a new symbol) with which one can transform x to y. Edit distance finds applications in computational biology and natural language processing, e.g., the correction of spelling mistakes or OCR errors, and approximate string matching, where the objective is to find matches for short strings in many longer texts, in situations where a small number of differences is to be expected. If $|x| \leq n$, $|y| \leq k$, the edit distance is solvable in time $O(nk)$ by dynamic programming [12]. If $n = k$, then, under the strong exponential time hypothesis (SETH), there is no classical algorithm computing edit distance in time $O(n^{2-\epsilon})$ for $\epsilon > 0$ [13] and the dynamic programming algorithm is essentially optimal.

However, SETH does not apply to quantum algorithms. Namely, SETH asserts that there is no algorithm for general instances of SAT that is substantially better than a naive search. Quantumly, simple use of Grover's search gives a quadratic advantage over naive search. This leads to the question: can this quadratic advantage be extended to edit distance (and other problems that have lower bounds based on SETH)?

Since edit distance is quite important in classical algorithms, the question about its quantum complexity has attracted substantial interest from various researchers. Boroujeni et al. [14] invented a better-than-classical quantum algorithm for approximating the edit distance, which was later superseded by a better classical algorithm of [15]. However, there have been no quantum algorithms computing the edit distance exactly (which is the most important case).

The main idea of the classical algorithm for edit distance is as follows:

- We construct a weighted version of the directed 2D grid (with edge weights 0 and 1) that encodes the edit distance problem for strings x and y, with the edit distance being equal to the length of the shortest directed path from $(0, 0)$ to (n, k).
- We solve the shortest path problem on this graph and obtain the edit distance.

As a first step, we can study the question of whether the shortest path is of length 0 or more than 0. Then, we can view edges of length 0 as present and edges of length 1 as absent. The question "Is there a path of the length of 0?" then becomes "Is there a path from $(0, 0)$ to (n, k) in which all edges are present?". A lower bound for this problem would imply a similar lower bound for the shortest path problem and a quantum algorithm for it may contain ideas that would be useful for a shortest path quantum algorithm.

1.4 Our results

We use our lower bound on the balanced parentheses language to show an $\Omega(n^{1.5-\epsilon})$ lower bound for the connectivity problem on the directed 2D grid. This shows a limit on quantum algorithms for finding edit distance through the reduction to shortest paths. More generally, for an $n \times k$ grid ($n > k$), our proof gives a lower bound of $\Omega((\sqrt{n}k)^{1-\epsilon})$.

A trivial query upper bound is $O(nk)$, since there are $O(nk)$ variables in total. We show a nontrivial quantum algorithm when k is small, i.e., we show that the connectivity problem can be solved with $O(\sqrt{n}\log^{k/2} n)$ quantum queries.[1] This bound becomes trivial when $k = \Omega(\frac{\log n}{\log\log n})$. We also present another algorithm that has query complexity $O(\sqrt{nk\mathcal{S}}\log n)$, where $\mathcal{S}$ is the number of segments of connected

[1] Aaronson et al. [7] also give a bound of $O(\sqrt{n}\log^{m-1} n)$ but in this case m is the rank of the syntactic monoid, which can be exponentially larger than k.

edges in the grid that cannot be extended. Since $n \le \mathcal{S} \le nk$, this complexity can be more or less effective. It varies from $O(k\sqrt{n}\log n)$ to $O(kn\log n)$.

For the undirected 2D grid, we show a lower bound of $\Omega((nk)^{1-\epsilon})$, whenever $k \ge \log n$. Thus, the naive algorithm is almost optimal in this case. We also extend both of these results to higher dimensions, obtaining a lower bound of $\Omega((n_1 n_2 \dots n_d)^{1-\epsilon})$ for an undirected $n_1 \times n_2 \times \cdots \times n_d$ grid in d dimensions and a lower bound of $\Omega(n^{(d+1)/2-\epsilon})$ for a directed $n \times n \times \cdots \times n$ grid in d dimensions.

In a recent work, an $\Omega(n^{1.5})$ lower bound for edit distance was shown by Buhrman et al. [10], assuming a quantum version of the Strong Exponential Time Hypothesis (QSETH). As part of this result, they give an $\Omega(n^{1.5})$ query lower bound for a different path problem on a 2D grid. Then, QSETH is invoked to prove that no quantum algorithm can be faster than the best algorithm for this shortest path problem. Neither of the two results follows directly one from another, as different shortest path problems are used.

The algorithms presented in the paper use nested Grover search algorithm in a recursive way, which requires manipulation of a large-scale fault-tolerant quantum computer. Our algorithms are thus far from practical at the current stage. In the meantime, recent developments in quantum error correction [16] and the announcement of a 400+ qubits quantum processor [17] give us hope for close future progress in this area.

2 Definitions

Let Σ be an alphabet. For a word $x \in \Sigma^*$ and a symbol $a \in \Sigma$, let $|x|_a$ be the number of occurrences of a in x. Here Σ^* is the set of all strings of any length of symbols from Σ.

For two (possibly partial) Boolean functions $g : G \to \{0,1\}$, where $G \subseteq \{0,1\}^n$, and $h : H \to \{0,1\}$, where $H \subseteq \{0,1\}^m$, we define the composed function $g \circ h : D \to \{0,1\}$, with $D \subseteq \{0,1\}^{nm}$, as $(g \circ h)(x) = g\left(h(x_1, \dots, x_m), \dots, h(x_{(n-1)m+1}, \dots, x_{nm})\right)$. Given a Boolean function f and a nonnegative integer d, we define f^d recursively as f iterated d times: $f^d = f \circ f^{d-1}$ with $f^1 = f$.

For a matrix Γ, $\|\Gamma\|$ denotes the spectral norm of Γ: $\|\Gamma\| = \max_{\overrightarrow{x} \neq 0} \frac{\|\Gamma \overrightarrow{x}\|}{\|\overrightarrow{x}\|}$ where $\|\overrightarrow{x}\|$ is the 2-norm of a vector.

2.1 Quantum query model

We use the standard form of the quantum query model. Let $f : D \to \{0,1\}$, $D \subseteq \{0,1\}^n$ be an n variable function we wish to compute on an input $x \in D$. We have oracle access to the input x—it is realized by a specific unitary transformation usually defined as $t|i\rangle t|z\rangle t|w\rangle \to t|i\rangle t|z + x_i \pmod 2\rangle t|w\rangle$ where the $t|i\rangle$ register indicates the index of the variable we are querying, $t|z\rangle$ is the output register, and $t|w\rangle$ is some auxiliary work-space. The operation is implemented by the CNOT gate. An algorithm in the query model consists of alternating applications of arbitrary unitaries independent of the input and the query unitary, and measurement in the end. The smallest number of queries for an algorithm that outputs $f(x)$ with probability $\geq \frac{2}{3}$ on all x is called the quantum query complexity of the function f and is denoted by $Q(f)$.

We refer the readers to [18–20] for more details on quantum computing and [5] for recent researches on quantum query model.

Let a symmetric matrix Γ be called an adversary matrix for f if the rows and columns of Γ are indexed by inputs $x \in D$ and $\Gamma_{xy} = 0$ if $f(x) = f(y)$. Let $\Gamma^{(i)}$ be a similarly sized matrix such that $\Gamma^{(i)}_{xy} = \begin{cases} \Gamma_{xy} & \text{if } x_i \neq y_i \\ 0 & \text{otherwise} \end{cases}$. Then, let $\mathrm{Adv}^{\pm}(f) = \max\limits_{\substack{\Gamma \text{ - an adversary} \\ \text{matrix for } f}} \dfrac{\|\Gamma\|}{\max_i \|\Gamma^{(i)}\|}$ be called the adversary bound and let $\mathrm{Adv}(f) = \max\limits_{\substack{\Gamma \text{ - an adversary matrix for } f \\ \Gamma \text{ - nonnegative}}} \dfrac{\|\Gamma\|}{\max_i \|\Gamma^{(i)}\|}$ be called the positive adversary bound. The following facts will be relevant to us:

$\mathrm{Adv}(f) \leq \mathrm{Adv}^{\pm}(f)$; $Q(f) = \Theta(\mathrm{Adv}^{\pm}(f))$ [21]; $\mathrm{Adv}^{\pm}$ composes exactly even for partial Boolean functions f and g, meaning, $\mathrm{Adv}^{\pm}(f \circ g) = \mathrm{Adv}^{\pm}(f) \cdot \mathrm{Adv}^{\pm}(g)$ [22, Lemma 6].

2.2 Reductions

We will say that a Boolean function f is reducible to g and denote it by $f \leq g$ if there exists an algorithm that given an oracle O_x for an input of f transforms it into an oracle O_y for g using at most $O(1)$ calls of oracle O_x such that $f(x)$ can be computed from $g(y)$. Therefore, from $f \leq g$ we conclude that $Q(f) \leq Q(g)$ because one can compute $f(x)$ using the algorithm for $g(y)$ and the reduction algorithm that maps x to y.

2.3 Dyck languages of bounded depth

Let Σ be an alphabet consisting of two symbols: (and). The Dyck language L consists of all $x \in \Sigma^*$ that represent a correct sequence of opening and closing parentheses. We consider languages L_k consisting of all words $x \in L$ where the number of opening parentheses that are not closed yet never exceeds k. The language L_k corresponds to a query problem $\textsc{Dyck}_{k,n}(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n \in \{0, 1\}$ describe a word of length n in the natural way: the ith symbol of x is (if $x_i = 0$ and) if $x_i = 1$. $\textsc{Dyck}_{k,n}(x) = 1$ iff the word x belongs to L_k. For all $x \in \{0, 1\}^n$, we define $f(x) = |x|_0 - |x|_1$,

we call it the **balance**. We define a $+k$-substring (resp. $-k$-substring) as a substring whose balance is equal to k (resp. equal to $-k$). A $\pm k-$substring is a substring whose balance is equal to k in absolute value. For all $0 \leq i \leq j \leq n-1$, we define $x[i, j] = x_i, x_{i+1}, \ldots, x_j$. Finally, we define $h(x) = \max_{0 \leq i \leq n-1} f(x[0, i])$ and $h^-(x) = \min_{0 \leq i \leq n-1} f(x[0, i])$. We also define the function sign such that $\mathrm{sign}(a) = 1$ if $a > 0$, and $\mathrm{sign}(a) = -1$ if $a < 0$, $\mathrm{sign}(a) = 0$ if $a = 0$. A substring $x[i, j]$

is *minimal* if it does not contain a substring $x[i', j']$ such that $(i, j) \neq (i', j')$, and $f(x[i', j']) = f(x[i, j])$.

2.4 Connectivity on a directed 2D grid

Let $G_{n,k}$ be a directed version of an $n \times k$ grid in two dimensions, with vertices $(i, j), i \in \{0, 1, \ldots, n\}, j \in \{0, 1, \ldots, k\}$ and directed edges from (i, j) to $(i + 1, j)$ (if $i < n$) and from (i, j) to $(i, j + 1)$ (if $j < k$). If G is a subgraph of $G_{n,k}$, we can describe it by variables x_e corresponding to edges e of $G_{n,k}$: $x_e = 1$ if the edge e belongs to G and $x_e = 0$ otherwise. We consider a problem 2D- DCONNECTIVITY in which one has to determine whether G contains a path from $(0, 0)$ to (n, k): 2D- DCONNECTIVITY$_{n,k}(x_1, \ldots, x_m) = 1$ (where m is the number of edges in $G_{n,k}$) iff such a path exists.

2.5 Connectivity on an undirected 2D grid

Let $G_{n,k}$ be an undirected $n \times k$ grid, and let G be a subgraph of $G_{n,k}$. We describe G by variables x_e in a similar way and define 2D- CONNECTIVITY$_{n,k}(x_1, \ldots, x_m) = 1$ iff G contains a path from $(0, 0)$ to (n, k). We also consider d dimensional versions of these two problems, on $n_1 \times n_2 \times \ldots n_d$ grids. In the directed version (dD- DCONNECTIVITY), we have a subgraph G of a directed grid (with edges directed in the directions from $(0, \ldots, 0)$ to $(n_1, \ldots, n_d)$) and dD- DCONNECTIVITY$(x_1, \ldots, x_m) = 1$ iff G contains a directed path from $(0, \ldots, 0)$ to $(n_1, \ldots, n_d)$. The undirected version is defined similarly, with an undirected grid instead of a directed one.

3 A quantum algorithm for membership testing of DYCK$_{k,n}$

In this section, we give a quantum algorithm for DYCK$_{k,n}(x)$, where k can be a function of n. The general idea is that DYCK$_{k,n}(x) = 0$ if and only if one of the following conditions holds:

(i) x contains a $+(k + 1)$-substring;
(ii) x contains a substring $x[0, i]$ such that the balance $f(x[0, i]) = -1$;
(iii) the balance of the entire word $f(x) \neq 0$.

The main algorithm is presented in Sect. 3.2. It is based on a subroutine presented in Sect. 3.1.

3.1 $\pm k$-Substring search algorithm

The goal of this section is to describe a quantum algorithm that searches for a substring $x[i, j]$ that has a balance $f(x[i, j]) \in \{+k, -k\}$ for some integer k. Throughout this section, we find and consider only **minimal** substrings. A substring is minimal if it does not contain a proper substring with the same balance. Throughout this section, we use the following easily verifiable facts:

- For any two minimal $\pm k$-substrings $x[i, j]$ and $x[k, l]$: $i < k \implies j < l$. This induces a natural linear order among all $\pm k$-substrings according to their starting (or, equivalently, ending) positions.
- Minimal $+k$-substrings do not intersect with minimal $-k$-substrings.
- If $x[l_1, r_1]$ and $x[l_2, r_2]$ with $l_1 < l_2$ are two **consecutive** minimal $(k-1)$-substrings and their signs are the same, then $x[l_1, r_2]$ is a k-substring with this sign.

This algorithm is the basis of our algorithms for $\textsc{Dyck}_{k,n}$. The algorithm works recursively. It searches for two consecutive minimal $\pm(k-1)$-substrings $x[l_1, r_1]$ and $x[l_2, r_2]$ such that they either overlap or there are no $\pm(k-1)$-substrings between them. If both substrings $x[l_1, r_1]$ and $x[l_2, r_2]$ are $+(k-1)$-substrings, then we get a minimal $+k$-substring in total. If both substrings are $-(k-1)$-substrings, then we get a minimal $-k$-substring in total.

Our algorithm utilizes three subroutines. The first one is $\textsc{FindAtLeftmost}_k(l, r, t, d, s)$ which accepts as inputs: the borders l and r, where l and r are integers such that $0 \le l \le r \le n-1$; a position $t \in \{l, \dots, r\}$; a maximal length d for the substring, where d is an integer such that $0 < d \le r - l + 1$; the sign of the balance $s \subseteq \{+1, -1\}$. $+1$ is used for searching for a $+k$-substring, -1 is used for searching for a $-k$-substring, $\{+1, -1\}$ is used for searching for both. It outputs a triple (i, j, σ) such that $l \le i \le t \le j \le r$, $j - i + 1 \le d$, $f(x[i, j]) \in \{+k, -k\}$ and $\sigma = \mathrm{sign}(f(x[i, j])) \in s$. The substring should be the leftmost one that contains t, i.e., there is no other minimal $x[i', j']$ such that $i' < i$, $t \in [i', j']$, $f(x[i', j']) = f(x[i, j])$. If no such substrings have been found, the algorithm returns NULL.

The second one is $\textsc{FindAtRightmost}_k$. It is similar to the $\textsc{FindAtLeftmost}_k$, but finds the rightmost $\pm k$-substring, i.e., there is no other minimal $x[i', j']$ such that $j' > j$, $t \in [i', j']$, $f(x[i', j']) = f(x[i, j])$

The third one is $\textsc{FindFirst}_k(l, r, s, direction)$ and accepts as inputs: the borders l and r, where l and r are integers such that $0 \le l \le r \le n-1$; the sign of the balance $s \subseteq \{+1, -1\}$. a $direction \in \{left, right\}$. When the direction is right (respectively left), $\textsc{FindFirst}_k$ finds the first $\pm k$-substring from the left to the right (respectively from the right to the left) in $[l, r]$ of sign s.

These three subroutines are interdependent since $\textsc{FindAtLeftmost}_k$ uses $\textsc{FindFirst}_{k-1}$ and $\textsc{FindAtRightmost}_{k-1}$ as subroutines, $\textsc{FindFirst}_k$ uses $\textsc{FindAtLeftmost}_k$ and $\textsc{FindAtRightmost}_k$ as subroutines. A description of $\textsc{FindAtLeftmost}_k(l, r, t, d, s)$ follows. The algorithm is presented in Algorithm 1.

The description of the subroutine $\textsc{FindAtRightmost}_k(l, r, t, d, s)$ is similar and is omitted.

When $k = 2$, the procedure $\textsc{FindAtLeftmost}_2(l, r, t, d, s)$ checks that $x_t = x_{t-1}$ and $\mathrm{sign}(f(x[t-1, t])) \in s$. If yes, it has found the substring. Otherwise, it checks if $x_t = x_{t+1}$ and $\mathrm{sign}(f(x[t, t+1])) \in s$. If both checks fail, the procedure returns NULL. For $k > 2$, the procedure is the following.

Step 1 Check whether t is inside a $\pm(k-1)$-substring of length at most $d-1$, i.e., $v = (i, j, \sigma) \leftarrow \textsc{FindAtLeftmost}_{k-1}(l, r, t, d-1, \{+1, -1\})$. If $v \ne$ NULL, then $(i_1, j_1, \sigma_1) \leftarrow (i, j, \sigma)$ and the algorithm goes to Step 2. Otherwise, the algorithm goes to Step 6.

Algorithm 1 FINDATLEFTMOST$_k(l, r, t, d, s)$.

```
v = (i_1, j_1, σ_1) ← FINDATLEFTMOST_{k−1}(l, r, t, d − 1, {+1, −1})
if v ≠ NULL then                                   ▷ if t is inside a ±(k − 1)-substring
   v' = (i_2, j_2, σ_2) ← FINDATRIGHTMOST_{k−1}(l, r, i_1 − 1, d − 1, {+1, −1})
   if v' = NULL then
      v' = (i_2, j_2, σ_2) ← FINDFIRST_{k−1}(min(l, j_1 − d + 1), i_1 − 1, {+1, −1}, left)
   end if
   if v' ≠ NULL and σ_2 ≠ σ_1 then
      v' ← NULL
   end if
   if v' = NULL then
      v' = (i_2, j_2, σ_2) ← FINDATLEFTMOST_{k−1}(l, r, j_1 + 1, d − 1, {+1, −1})
      if v' = NULL then
         v' = (i_2, j_2, σ_2) ← FINDFIRST_{k−1}(j_1 + 1, min(i_1 + d − 1, r), {+1, −1}, right)
      end if
   end if
   if v' = NULL then
      return NULL
   end if
else
   v = (i_1, j_1, σ_1) ← FINDFIRST_{k−1}(t, min(t + d − 1, r), {+1, −1}, right)
   if v = NULL then
      return NULL
   end if
   v' = (i_2, j_2, σ_2) ← FINDFIRST_{k−1}(max(l, t − d + 1), t), {+1, −1}, left)
   if v' = NULL then
      return NULL
   end if
end if
if σ_1 = σ_2 and σ ∈ s and max(j_1, j_2) − min(i_1, i_2) + 1 ≤ d then
   return (min(i_1, i_2), max(j_1, j_2), σ_1)
else
   return NULL
end if
```

Step 2 Check whether $i_1 - 1$ is inside a $\pm(k-1)$-substring of length at most $d-1$ and choose the rightmost one: $v = (i, j, \sigma) \leftarrow$ FINDATRIGHTMOST$_{k-1}(l, r, i_1 - 1, d-1, \{+1, -1\})$. If $v =$ NULL, then the algorithm goes to Step 3. If $v \neq$ NULL and $\sigma = \sigma_1$, then $(i_2, j_2, \sigma_2) \leftarrow (i, j, \sigma)$ and go to Step 8. Otherwise, go to Step 4.

Step 3 Search for the first $\pm(k-1)$-substring on the left from $i_1 - 1$ at distance at most d, i.e., $v = (i, j, \sigma) \leftarrow$ FINDFIRST$_{k-1}(\min(l, j_1 - d + 1), i_1 - 1), \{+1, -1\}, left)$. If $v \neq$ NULL and $\sigma_1 = \sigma$, then $(i_2, j_2, \sigma_2) \leftarrow (i, j, \sigma)$ and go to Step 8. Otherwise, go to Step 4.

Step 4 Check whether $j_1 + 1$ is inside a $\pm(k-1)$-substring of length at most $d-1$, i.e., $v = (i, j, \sigma) \leftarrow$ FINDATLEFTMOST$_{k-1}(l, r, j_1 + 1, d - 1, \{+1, -1\})$. If $v \neq$ NULL, then $(i_2, j_2, \sigma_2) \leftarrow (i, j, \sigma)$ and go to Step 8. Otherwise, go to Step 5.

Step 5 Search for the first $\pm(k-1)$-substring on the right from $j_1 + 1$ at distance at most d, i.e., $v = (i, j, \sigma) \leftarrow$ FINDFIRST$_{k-1}(j_1 + 1, \min(i_1 + d -$

$1, r), \{+1, -1\}, right)$. If $v \neq$ NULL, then $(i_2, j_2, \sigma_2) \leftarrow (i, j, \sigma)$, then go to Step 8. Otherwise, return NULL.

Step 6 Search for the first $\pm(k-1)$-substring on the right at distance at most d from t, i.e., $v = (i, j, \sigma) \leftarrow \textsc{FindFirst}_{k-1}(t, \min(t + d - 1, r), \{+1, -1\}, right)$ If $v \neq$ NULL, then $(i_1, j_1, \sigma_1) \leftarrow (i, j, \sigma)$ and go to Step 7. Otherwise, returns NULL.

Step 7 Search for the first $\pm(k-1)$-substring on the left from t at distance at most d, i.e., $v = (i, j, \sigma) \leftarrow \textsc{FindFirst}_{k-1}(\max(l, t - d + 1), t), \{+1, -1\}, left)$ If $v \neq$ NULL, then $(i_2, j_2, \sigma_2) \leftarrow (i, j, \sigma)$ and go to Step 8. Otherwise, returns NULL.

Step 8 If $\sigma_1 = \sigma_2$, $\sigma_1 \in s$ and $\max(j_1, j_2) - \min(i_1, i_2) + 1 \leq d$, the subroutine returns $(\min(i_1, i_2), \max(j_1, j_2), \sigma_1)$, otherwise returns NULL.

By construction and induction on k, the two $\pm(k-1)$-substrings $x[i_1, j_1]$ and $x[i_2, j_2]$ (if they exist) involved in the procedure $\textsc{FindAtLeftmost}_k$ are always consecutive and minimal. $\textsc{FindAtLeftmost}_k$ thus returns a $\pm k$-substring, if both substrings have the same sign.

Using this basic procedure, we then search for a $\pm k-$substring by searching for a t and d such that $\textsc{FindAtLeftmost}_k(l, r, t, d, s)$ returns a non-NULL value. Unfortunately, our algorithms have two-sided bounded error: they can, with small probability, return NULL even if a substring exists or return a wrong substring instead of NULL. In this setting, Grover's search algorithm is not directly applicable, and we need to use a more sophisticated search [23]. Furthermore, simply applying the search algorithm naively does not give the right complexity. Indeed, if we search for a substring of length roughly d (say between d and $2d$), we can find one with expected running time $O(\sqrt{(r-l)/d})$ because at least d values of t will work. On the other hand, if there are no such substrings, the expected running time will be $O(\sqrt{r-l})$. Intuitively, we can do better because if there is a substring of length at least d then there are at least d values of t that work. Hence, we only need to distinguish between no solutions, or at least d. This allows us to stop the Grover iteration early and make $O(\sqrt{(r-l)/d})$ queries in all cases.

Lemma 1 (Modified from [23]) *Given n algorithms, quantum or classical, each computing some bit-value with bounded error probability, and some $T \geq 1$, there is a quantum algorithm that uses $O(\sqrt{n/T})$ queries and with constant probability: returns the index of a "1", if there are at least T "1s" among the n values; returns* NULL *if there are no "1"; returns anything otherwise.*

Proof The main loop of the algorithm of [23] is the following, assuming the algorithms have an error at most $1/9$:

- for $m = 0$ to $\lceil \log_9 n \rceil$-1 do:
  1. Run A_m 1000 times,
  2. Verify the 1000 measurements, each by $O(\log n)$ runs of the corresponding algorithm,
  3. If a solution has been found, then output a solution and stop
- Output "no solutions"

The key of the analysis is that if the (unknown) number t of solutions lies in the interval $[n/9^{m+1}, n/9^m]$, then A_m succeeds with constant probability. In all cases, if there are no solutions, A_m will never succeed with high probability (ie the algorithm only applies good solutions).

In our case, we allow the algorithm to return anything (including NULL) if $t < T$. This means that we only care about the values of m such that $n/9^m \geq T$, that is $m \leq \log_9 \frac{n}{T}$. Hence, we simply run the algorithm with this new upper bound for d, and it will satisfy our requirements with constant probability. The complexity is

$$\sum_{m=0}^{\left\lfloor log_9 \frac{n}{T} \right\rfloor} 1000 \cdot O(3^m) + 1000 \cdot O(\log n) = O(3^{\log_9 \frac{n}{T}}) = O(\sqrt{n/T}). \qquad \square$$

The algorithm that uses the above ideas is presented in Algorithm 2.

Algorithm 2 FindFixedLen$_k(l, r, d, s)$. Search for any $\pm k$-substring of length $\in [d/2, d]$

Find t such that $v_t \leftarrow$ FindAtLeftmost$_k(l, r, t, d, s) \neq$ NULL using Lemma 1 with $T = d/2$.
return v_t or NULL if none.

We can then write an algorithm FindAny$_k(l, r, s)$ that searches for any $\pm k$-substring. We consider a randomized algorithm that uniformly chooses a of power 2 from $[2^{\lceil \log_2 k \rceil}, (r-l)]$, i.e., $d \in \{2^{\lceil \log_2 k \rceil}, 2^{\lceil \log_2 k \rceil + 1}, \dots, 2^{\lceil \log_2 (r-l) \rceil}\}$. For the chosen d, we run Algorithm 2. So, the algorithm will succeed with probability at least $O(1/\log(r-l))$. We can apply Amplitude amplification and ideas from Lemma 1 to this and get an algorithm that uses $O(\sqrt{\log(r-l)})$ iterations.

Algorithm 3 FindAny$_k(l, r, s)$. Search for any $\pm k$-substring

Find $d \in \{2^{\lceil \log_2 k \rceil}, 2^{\lceil \log_2 k \rceil + 1}, \dots, 2^{\lceil \log_2 (r-l) \rceil}\}$ such that:
$v_d \leftarrow$ FindFixedLen$_k(l, r, d, s) \neq$ NULL using amplitude amplification.
return v_d or NULL if none.

Finally, we present the algorithm that finds the first $\pm k$-substring—FindFirst$_k$. Let us consider the case $direction = right$. We first find the smallest segment from the left to the right such that its length w is a power of 2 and it contains a $\pm k$-substring. We do so by doubling the length of the segment until we find a $\pm k$-substring. We now have a segment that contains a $\pm k$-substring, and we want to find the leftmost one. We do so by the following variant of binary search. At each step, let $mid = \lfloor (lBorder + rBorder)/2 \rfloor$ be the middle of the search segment $[lBorder, rBorder]$. There are three cases:

- There is a k-substring in $[lBorder, mid]$, then the leftmost k-substring is in this segment.
- There are no k-substrings in $[lBorder, mid]$, but mid is inside a k-substring. Then, the leftmost k-substring that contains mid is the required substring.

- There are no k-substrings in $[lBorder, mid]$ and mid is not inside a k-substring. Then, the required substring is in $[mid + 1, rBorder]$.

In each iteration of the loop, the algorithm halves the search space or finds the first k-substring itself if it contains mid. If $direction = left$, we replace FINDATLEFTMOST$_k$ by FINDATRIGHTMOST$_k$ that finds the rightmost $\pm k$-substring that containts mid.

Let us present a detailed description of this algorithm. The FINDFIRST$_k$ procedure calls FINDLEFTFIRST$_k$ or FINDRIGHTFIRST$_k$ depending on the direction. Since both versions are essentially symmetric, we only present the search from the left below (i.e., when the direction is right). For reasons that become clear in the proof, we need to boost the success probability of some calls. We do so by repeating them several times and taking the majority: by this, we mean that we take the most common answer, and return an error in case of a tie.

Algorithm 4 FINDRIGHTFIRST$_k(l, r, s)$. The algorithm for searching for the first $\pm k$-substring

```
lBorder ← l, rBorder ← r
d ← 1                                                         ▷ depth of the search
while lBorder + 1 < rBorder do
    mid ← ⌊(lBorder + rBorder)/2⌋
    v_l ← FINDANY_k(lBorder, mid, s)             ▷ repeat 2d times and take the majority
    if v_l ≠ NULL then
        rBorder ← mid
    end if
    if v_l = NULL then
        v_mid ← FINDFIXEDPOS_k(lBorder, rBorder, mid, s, left)       ▷ majority of 2d runs
        if v_mid ≠ NULL then
            v ← v_mid
            Stop the loop.
        end if
        if v_mid = NULL then
            lBorder ← mid + 1
        end if
    end if
    d ← d + 1
end while
return v
```

Proposition 2 *For any $\varepsilon > 0$ and k, algorithms* FINDATLEFTMOST$_k$, FINDFIXEDLEN$_k$, FINDANY$_k$ *and* FINDFIRST$_k$ *have two-sided error probability $\varepsilon < 0.5$ and return, when correct:*

- *If t is inside a $\pm k-$substring of sign s of length up to d in $x[l, r]$, then* FINDATLEFTMOST$_k$ *will return such a substring, otherwise, it returns* NULL. *The running time is $O(\sqrt{d}(\log(r - l))^{0.5(k-2)})$.*
- FINDFIXEDLEN$_k$ *either returns a $\pm k-$substring of sign s and length at most d in $x[l, r]$, or* NULL. *It is only guaranteed to return a substring if there exists $\pm k-$substring of length at least $d/2$, otherwise, it can return* NULL. *The running time is $O(\sqrt{r - l}(\log(r - l))^{0.5(k-2)})$.*

- FINDANY$_k$ *returns any* $\pm k-$*substring of sign* s *in* $x[l, r]$, *otherwise it returns* NULL. *The running time is* $O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})$.
- FINDFIRST$_k$ *returns the first* $\pm k-$*substring of sign* s *in* $x[l, r]$ *in the specified direction, otherwise it returns* NULL. *The running time is* $O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})$.

Proof We prove the result by induction on k. The base case of $k = 2$ is obvious because of the simplicity of FINDATLEFTMOST$_2$ and FINDATRIGHTMOST$_2$ procedures. We first prove the correctness of all the algorithms, assuming there are no errors. In the end, we explain how to deal with the errors.

We start with FINDATLEFTMOST$_k$: there are different cases to be considered when searching for a $+k$-substring $x[i, j]$ of length $\leq d$.

1. Assume that there are j_1 and i_2 such that $i < j_1 < i_2 < j$, $|f(x[i, j_1])| = |f(x[i_2, j])| = k-1$ and $\text{sign}(f(x[i, j_1])) = \text{sign}(f(x[i_2, j])) \in s$. If $t \in \{i_2, \dots, j\}$, then the algorithm finds $x[i_2, j]$ in Step 1 and the first invocation of FINDFIRST$_{k-1}$ in Step 3 finds $x[i, j_1]$. If $t \in \{i, \dots, j_1\}$, then the algorithm finds $x[i, j_1]$ in Step 1 and the second invocation of FINDFIRST$_{k-1}$ in Step 5 finds $x[i_2, j]$. If $j_1 < t < i_2$, then the third invocation of FINDFIRST$_{k-1}$ in Step 6 finds $x[i_2, j]$ and the fourth invocation of FINDFIRST$_{k-1}$ in Step 7 finds $x[i, j_1]$.
2. Assume that there are j_1 and i_2 such that $i < i_2 < j_1 < j$, $|f(x[i, j_1])| = |f(x[i_2, j])| = k-1$ and $\text{sign}(f(x[i, j_1])) = \text{sign}(f(x[i_2, j])) \in s$. If $t \in \{i, \dots, j_1\}$, then the algorithm finds $x[i, j_1]$ in Step 1. After that, it finds $x[i_2, j]$ in Step 4. If $t \in \{j_1 + 1, \dots, j\}$, then the algorithm finds $x[i_2, j]$ in Step 1. After that, it finds $x[i, j_1]$ in Step 2.

By induction, the running time of each FINDATLEFTMOST$_{k-1}$ invocation is $O(\sqrt{d}$ $(\log(r-l))^{0.5(k-3)})$, and the running time of each FINDFIRST$_{k-1}$ invocation is $O(\sqrt{d}(\log(r-l))^{0.5(k-2)})$.

We now look at FINDFIXEDLEN$_k$: by construction and definition of FINDATLEFTMOST$_k$, if the algorithm returns a value, it is a valid substring (with high probability). If there exists a substring of length at least $d/2$, then any query to FINDATLEFTMOST$_k$ with a value of t in this interval will succeed; hence, there are at least $d/2$ solutions. Therefore, by Lemma 1, the algorithm will find one with high probability and make $O\left(\sqrt{\frac{r-l}{d/2}}\right)$ queries. Each query has complexity $O(\sqrt{d}(\log(r-l))^{0.5(k-2)})$ by the previous paragraph; hence, the running time is bounded by $O(\sqrt{r-l}(\log(r-l)^{0.5(k-2)})$.

We can now analyze FINDANY$_k$ Assume that the shortest $\pm k$-substring $x[i, j]$ is of length $g = j - i + 1$. Therefore, there is a d such that $d \leq g \leq 2d$ and the FINDFIXEDLEN$_k$ procedure returns a substring for this d with constant success probability. So, the success probability of the randomized algorithm is at least $O(1/\log(l-r))$. Therefore, the amplitude amplification does $O(\sqrt{\log(r-l)})$ iterations. The running time of FINDFIXEDLEN$_k$ is $O(\sqrt{r-l}(\log(r-l))^{0.5(k-2)})$ by induction; hence, the total running time is: $O(\sqrt{r-l}(\log(r-l))^{0.5(k-2)}\sqrt{\log(l-r)}) = O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})$.

Finally, we analyze FINDFIRST$_k$: Let us prove the correctness of the algorithm for direction $=$ right and $s = \{+1\}$. The proof for other parameters is similar.

First, we show the correctness of the algorithm assuming there are no errors. The algorithm is essentially a binary search. At each step, we find the middle of the search segment [lBorder, rBorder] that is mid $= \lfloor(\text{lBorder} + \text{rBorder})/2\rfloor$. There are three options.

- There is a k-substring in [lBorder, mid], then the leftmost k-substring is in this segment.
- There are no k-substrings in [lBorder, mid], but mid is inside a k-substring. If we find the leftmost substring containing min, it is the required substring.
- There are no k-substrings in [lBorder, mid] and mid is not inside a k-substring. Then, the required substring is in [mid $+ 1$, rBorder].

In each iteration of the loop, the algorithm finds a smaller segment containing the leftmost k-substring or finds it if it contains mid. We find the k-substring in the iteration that corresponds to the [lBorder, rBorder] segment such that $(\text{rBorder} - \text{lBorder})/2 \leq j - i$ or earlier.

Second, we compute complexity of the algorithm (taking into account the repetitions and majority votes).

The u-th iteration of the loop considers a segment [lBorder, rBorder]. The length of this segment is at most $w \cdot 2^{-(u-1)}$ where $w = r - l$. The complexity of $\text{FINDANY}_k(\text{lBorder}, \text{mid}, s)$ is at most $O\Big(\sqrt{w \cdot 2^{-(u-1)-1}}\big(\log(w \cdot 2^{-(u-1)-1})\big)^{0.5(k-1)}\Big) = O\left(\sqrt{w \cdot 2^{-(u-1)-1}}\,(\log(r-l))^{0.5(k-1)}\right)$. Also, $\text{FINDFIXEDPOS}_k(\text{lBorder}, \text{rBorder}, \text{mid}, s, \text{left})$ has complexity $O\left(\sqrt{w \cdot 2^{-(u-1)}}\,\big(\log(w \cdot 2^{-(u-1)})\big)^{0.5(k-1)}\right) = O\left(\sqrt{w \cdot 2^{-(u-1)}}\,(\log(r-l))^{0.5(k-1)}\right)$. So the total complexity of the u-th iteration is $O\left(u\sqrt{w \cdot 2^{-(u-1)}}\,(\log(r-l))^{0.5(k-1)}\right)$, since at the u-th iteration, we repeat each call $2u$ times to take a majority. The number of iterations is at most $\log_2 w$. Let us compute the total complexity of the binary search part:

$$
\begin{aligned}
&O\left(\sum_{u=1}^{\log_2 w} 2u\sqrt{w \cdot 2^{-(u-1)}}\,(\log(r-l))^{0.5(k-1)}\right) \\
&\quad = O\left(\sqrt{w}\,(\log(r-l))^{0.5(k-1)} \sum_{u=1}^{\log_2 w} u(\sqrt{2})^{-(u-1)}\right) \\
&\quad = O\left(\sqrt{w}\,(\log(r-l))^{0.5(k-1)} \sum_{u=0}^{\infty}(u+1)(\sqrt{2})^{-u}\right) \\
&\quad = O\left(\sqrt{w}\,(\log(r-l))^{0.5(k-1)}\,\frac{\sqrt{2}^2}{(\sqrt{2}-1)^2}\right) \\
&\quad = O\left(\sqrt{w}\,(\log(r-l))^{0.5(k-1)}\right).
\end{aligned}
$$

Finally, we need to analyze the success probability of the algorithm: at the uth iteration, the algorithm will run each test $2u$ times and each test has a constant probability of failure ε. Hence for the algorithm to fail (that is make a decision that will not lead to the first $\pm k$-substring) at iteration u, at least half of the $2u$ runs must fail: this happens with probability at most

$$\binom{2u}{u}\varepsilon^u \le \left(\frac{2ue}{u}\right)^u \varepsilon^u \le (2e\varepsilon)^u.$$

Hence, the probability that the algorithm fails is bounded by:

$$\sum_{u=1}^{\log_2 w} (2e\varepsilon)^u \le \sum_{u=1}^{\infty}(2e\varepsilon)^u \le \frac{2e\varepsilon}{1-2e\varepsilon}.$$

By taking ε small enough (say $2e\varepsilon < \frac{1}{3}$), which is always possible by repeating the calls a constant number of times to boost the probability, we can ensure that the algorithm has a probability of failure less than $1/2$. An extended version of this proof technique is presented in [24].

We now turn to error analysis The case of $\textsc{FindAtLeftmost}_k$ is easy: the algorithm makes at most 5 recursive calls, each having a success probability of $1-\varepsilon$. Hence, it will succeed with probability $(1-\varepsilon)^5$. We can boost this probability to $1-\varepsilon$ by repeating this algorithm a constant number of times. Note that this constant depends on ε.

The analysis of $\textsc{FindFixedLen}_k$ follows from [23] and Lemma 1: since $\textsc{FindAtLeftmost}_k$ has two-sided error ε, there exists a search algorithm with two-sided error ε. □

3.2 The algorithm for $\textsc{Dyck}_{k,n}$

To solve $\textsc{Dyck}_{k,n}$, we modify the input x. As the new input, we use $x' = 1^k x 0^k$. $\textsc{Dyck}_{k,n}(x) = 1$ iff there are no $\pm(k+1)$-substrings in x'. This idea is presented in Algorithm 5.

Algorithm 5 $\textsc{Dyck}_{k,n}(x)$. The Quantum Algorithm for $\textsc{Dyck}_{k,n}$

$x \leftarrow 1^k x 0^k$
$v = \textsc{FindAny}_{(k+1)}(0, n+2k-1, \{+1,-1\})$
return $v ==$ NULL

Theorem 3 *Algorithm 5 solves* $\textsc{Dyck}_{k,n}$ *and the running time of Algorithm 5 is* $O(\sqrt{n}(\log n)^{0.5k})$. *The algorithm has two-side error probability* $\varepsilon < 0.5$.

Proof Let us show that if x' contains $\pm(k+1)$-substring, then one of three conditions of $\textsc{Dyck}_{k,n}$ problem is broken.

Assume that x' contains $(k+1)$ substring $x'[i,j]$. If $j \geq k+n$, then $f(x[i-k, n-1]) > 0$, because $f(x'[n,j]) = j-n+1 \leq k < k+1$. Therefore, prefix $x[0, i-k]$ is such that $f(x[0, i-k-1]) < 0$ or $f(x[0, n-1]) > 0$ because $f(x[0, n-1]) = f(x[0, i-k]) + f(x[i-k-1, n-1])$. So, in that case, we break one of the conditions of $\textsc{Dyck}_{k,n}$ problem.

If $j < k+n$, then $x[i-k, j-k]$ is $(k+1)$ substring of x.

Assume that x' contains $-(k+1)$ substring $x'[i,j]$. If $i < k$, then $f(x[0, j-k]) < 0$, because $f(x'[i, k-1]) = -(k-i) \geq -k > -(k+1)$ and $f(x[0, j-k]) = f(x'[k,j]) = f(x[i,j]) - f(x[i, k-1])$. So, in that case, the second condition of the $\textsc{Dyck}_{k,n}$ problem is broken.

The complexity of Algorithm 5 is the same as the complexity of $\textsc{FindAny}_{k+1}$ for x' that is $O(\sqrt{n+2k}(\log(n+2k))^{0.5k})$ due to Proposition 2.

We can assume $n \geq 2k$ (otherwise, we can update $k \leftarrow n/2$). Hence,

$$
\begin{aligned}
&O(\sqrt{n+2k}(\log(n+2k))^{0.5k})\\
&\quad = O(\sqrt{2n}(\log(2n))^{0.5k}) = O(\sqrt{n}(2\log n)^{0.5k})\\
&\quad = O(\sqrt{n}(\log n)^{0.5k})
\end{aligned}
$$

The error probability is the same as the complexity of $\textsc{FindAny}_{k+1}$. □

4 Lower bounds for Dyck languages

Theorem 4 *There exist constants $c_1, c_2 > 0$ such that $Q\left(\textsc{Dyck}_{c_1 \ell m, c_2 (2m)^\ell}\right) = \Omega\left(m^\ell\right)$.*

Proof We will use the partial Boolean function $\mathrm{EX}_m^{a|b} = \begin{cases} 1, & \text{if } |x|_0 = a \\ 0, & \text{if } |x|_0 = b. \end{cases}$ We prove the theorem by a reduction $\left(\mathrm{EX}_{2m}^{m|m+1}\right)^\ell \leq \textsc{Dyck}_{c_1 \ell m, c_2 (2m)^\ell}$.

Before we describe the reduction in detail, we sketch the main idea. Recall that $f(x) = |x|_0 - |x|_1$. Note that

$$
\begin{aligned}
\mathrm{EX}_{2m}^{m|m+1}(x) = 0 &\iff f(x) = 2\\
\mathrm{EX}_{2m}^{m|m+1}(x) = 1 &\iff f(x) = 0
\end{aligned}
$$

whereas

$$
\begin{aligned}
\textsc{Dyck}_{k,n}(x) = 1 \iff &\left(\max_{p-\text{ prefix of } x} f(p) \leq k\right) \wedge \left(\min_{p-\text{ prefix of } x} f(p) \geq 0\right)\\
&\wedge (f(x) = 0).
\end{aligned}
$$

If we could make sure that the minimum and maximum constraints are satisfied, $\textsc{Dyck}_{k,n}$ could be used to compute $\mathrm{EX}_{2m}^{m|m+1}$. To ensure the minimum constraint, we map each 0 to 00 and 1 to 01. However, this increases $f(x)$ by $2m$, which can be fixed by appending 1^{2m} at the end. Importantly, the resulting sequence x' has $f(x') = f(x)$. The first constraint (maximum over prefixes) can be fulfilled by having a sufficiently large k; $k = 2m + 3$ would suffice here. The same idea can be applied iteratively to $\mathrm{Ex}_{2m}^{m|m+1}$ where the inputs, which could now be the results of functions $\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell-1} = x_i$, have been recursively mapped to sequences x_i' with $f(x_i') = \begin{cases} 2 \text{ if } x_i = 0 \\ 0 \text{ if } x_i = 1 \end{cases}$.

The reduction formally is as follows.

We call a string $B \in \{0,1\}^w$ of even length a (w,h)-sized block with width w and height h iff for any prefix x of B: $0 \le f(x) \le h$ and either $f(B) = 0$ or $f(B) = 2$.

We establish a correspondence between inputs to $\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell}$ that satisfy the promise and (w,h)-sized blocks B for appropriately chosen w, h, so that $\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell} = 1$ iff $f(B) = 0$.

For $l = 0$ (the input bits), we have 0 corresponding to a $(2,2)$-sized block of 00 and 1 to a $(2,2)$-sized block of 01.

For $l > 0$, let us have input bits $x = (x_1, x_2, \ldots, x_{2m})$ of $\mathrm{Ex}_{2m}^{m|m+1}$ satisfying the input promise. Assume that the bits (that could be equal to values of $\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell-1}$) correspond to (w,h)-sized blocks $B_1, B_2, \ldots, B_{2m}$. Define the sequence $B' = B_1 B_2 \ldots B_{2m} 1^{2m}$. Then, it is easy to verify the following claims:

1. B' is a $(2m(w+1), 2(m+1)+h)$-sized block;
2. The output bit of $\mathrm{Ex}_{2m}^{m|m+1}(x)$ *corresponds* to B' because

$$f(B') = \sum_{i=1}^{2m} f(B_i) + f(1^{2m}) = \begin{cases} 2 & \text{if } \mathrm{Ex}_{2m}^{m|m+1}(x) = 0 \\ 0 & \text{if } \mathrm{Ex}_{2m}^{m|m+1}(x) = 1 \end{cases}.$$

For $l = 0$, the inputs correspond to $(2,2)$-sized blocks. Each level adds $2(m+1)$ to the height of the blocks reaching $2 + 2\ell(m+1) = O(m\ell)$. The width of blocks reaches $O((2m)^{\ell})$.

Since for all (w,h)-sized blocks B: $\textsc{Dyck}_{h,w}(B) = 1 \iff f(B) = 0$ one can solve the $\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell}$ problem by running $\textsc{Dyck}_{h,w}$ on the *corresponding* block.

See Fig. 1.

It is known that $Adv^{\pm}\left(\mathrm{Ex}_{2m}^{m|m+1}\right) \ge Adv\left(\mathrm{Ex}_{2m}^{m|m+1}\right) > m$ [25, Theorem 5.4]. The Adversary bound composes even for partial Boolean functions [22, Lemma 1], therefore $Q\left(\left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell}\right) = \Omega\left(m^{\ell}\right)$. Via the reduction the same bound applies to $\textsc{Dyck}_{c_1 \ell m, c_2 (2m)^{\ell}}$. □

Theorem 5 *For any $\epsilon > 0$, there exists $c > 0$ such that $Q\left(\textsc{Dyck}_{c \log n, n}\right) = \Omega\left(n^{1-\epsilon}\right)$.*

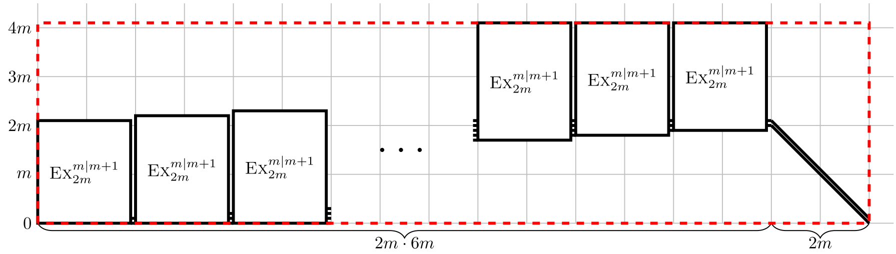


Fig. 1 The reduction $\mathrm{Ex}_{2m}^{m|m+1} \circ \mathrm{Ex}_{2m}^{m|m+1} \leq \mathrm{DYCK}_{4m+6,12m^2+2m}$. The line of the graph follows the input word along the x-axis and shows the number of yet-unclosed parenthesis along the y-axis (i.e., a zoomed-out version of Fig. 2). The input word $B_1 B_2 \ldots B_{2m} 1^{2m}$ corresponds to the outer function $\mathrm{Ex}_{2m}^{m|m+1}$ with B_j being a block corresponding to the output of an inner $\mathrm{Ex}_{2m}^{m|m+1}$. The ticks at the starts and ends of blocks depict that if the line enters the block at height i, it exits at height i or $i+2$. In the block, the line never goes below 0 or above $h+i$. The red dashed part then forms a new block B'. By replacing the blocks B_j with blocks B', we can further iterate $\mathrm{Ex}_{2m}^{m|m+1}$ to get the reduction $\mathrm{Ex}_{2m}^{m|m+1} \circ \left(\mathrm{Ex}_{2m}^{m|m+1}\right)^{\ell-1} \leq \mathrm{DYCK}_{O(\ell m), O\left((2m)^\ell\right)}$

Proof For any $\epsilon > 0$, there exists an m such that $Adv^{\pm}\left(\mathrm{Ex}_{2m}^{m|m+1}\right) \geq (2m)^{1-\epsilon}$. Without loss of generality, we may assume that $(2m)^\ell = n$. From Theorem 4 with $\ell = \log_{2m} n$, we obtain $c_2 (2m)^\ell = c_2 n$ and height $c_1 m \ell = \Theta(\log n)$. The query complexity is at least $\left((2m)^{1-\epsilon}\right)^\ell = \left((2m)^\ell\right)^{1-\epsilon} = n^{1-\epsilon}$. Therefore, $Q\left(\mathrm{DYCK}_{c \log n, n}\right) = \Omega\left(n^{1-\epsilon}\right)$. $\square$

For constant depths, the following bound can be derived:

Theorem 6 *There exists a constant* $c_1 > 0$ *such that* $Q(\mathrm{DYCK}_{c_1 \ell, n}) = \Omega(2^{\frac{\ell}{2}} \sqrt{n})$.

Proof Let $m = 4$ in Theorem 4. Then, $Q\left(\mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}\right) = \Omega\left(4^\ell\right)$ for some constants $c_1, c_2 > 0$. Consider the function $\mathrm{AND}_{\frac{n}{c_2 8^\ell}} \circ \mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}$ with a promise that AND_k has as an input either k or $k-1$ ones. Then,

$$Q\left(\mathrm{AND}_{\frac{n}{c_2 8^\ell}} \circ \mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}\right) = \Theta\left(\mathrm{Adv}^{\pm}\left(\mathrm{AND}_{\frac{n}{c_2 8^\ell}} \circ \mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}\right)\right) \quad \text{and}$$

$$\mathrm{Adv}^{\pm}\left(\mathrm{AND}_{\frac{n}{c_2 8^\ell}} \circ \mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}\right) \geq \mathrm{Adv}^{\pm}\left(\mathrm{AND}_{\frac{n}{c_2 8^\ell}}\right) \mathrm{Adv}^{\pm}\left(\mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}\right)$$
$$= \Omega\left(2^{\frac{\ell}{2}} \sqrt{n}\right),$$

with the second step following from the composition of $Adv^{\pm}$ for partial functions [22]. This implies the same lower bound on $\mathrm{DYCK}_{c_1 \ell, n}$ because the computation of the composition $\mathrm{AND}_{\frac{n}{c_2 8^\ell}} \circ \mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}$ can be straightforwardly reduced to $\mathrm{DYCK}_{c_1 \ell, n}$ by a simple concatenation of $\mathrm{DYCK}_{c_1 \ell, c_2 8^\ell}$ instances. $\square$

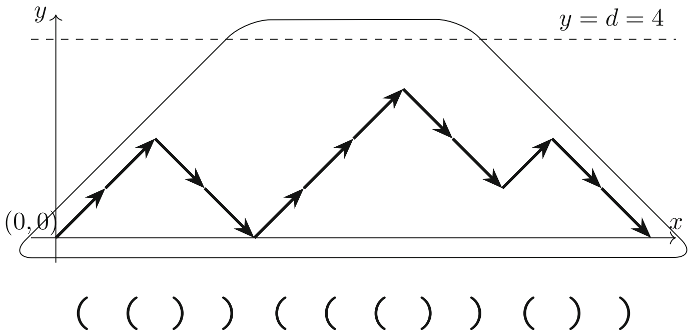


Fig. 2 Representation of the Dyck word "(( )) ((( )) ( ))"

5 Quantum complexity of ST-CONNECTIVITY in grids

5.1 Quantum complexity of 2D-DCONNECTIVITY$_{n,k}$

Theorem 7 *For any $n \geq k$ and $\epsilon > 0$, $Q(\text{2D- DCONNECTIVITY}_{n,k}) = \Omega\left((\sqrt{n}k)^{1-\epsilon}\right)$.*

In particular, if we have a square grid, then

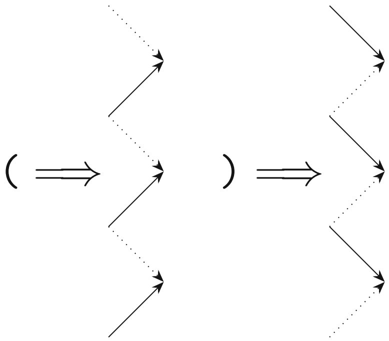
Corollary 8 *For any $\epsilon > 0$, $Q(\text{2D- DCONNECTIVITY}_{n,n}) = \Omega\left(n^{1.5-\epsilon}\right)$.*

Proof of Theorem 7 For any sequence w of m opening and closing parentheses, it is possible to plot the changes of depth, i.e., the number of opening parentheses minus the number of closing parentheses, for all prefixes of the sequence, see Fig. 2.

We can connect neighboring points by vectors $(1, 1)$ and $(1, -1)$ corresponding to the opening and closing parentheses, respectively. Clearly, $w \in L_d$ if and only if the path starting at the origin $(0, 0)$ ends at $(m, 0)$ and never crosses $y = 0$ and $y = d$. Consequently, a path corresponding to $w \in L_d$ always remains within the trapezoid bounded by $y = 0$, $y = d$, $y = x$, $y = -x + m$. This suggests a way of mapping DYCK$_{d,m}$ to the 2D- DCONNECTIVITY$_{n,k}$ problem:

1. An opening parenthesis in position i corresponds to a "column" of upwards sloping available edges $(i - 1, l) \to (i, l + 1)$ for all $l \in \{0, 1, \ldots, d - 1\}$ such that $i - 1 + l$ is even. A closing parenthesis in position i corresponds to downwards sloping available edges $(i - 1, l) \to (i, l - 1)$ for all $l \in \{1, \ldots, d\}$ such that $i - 1 + l$ is even. See Fig. 3.
2. The edges outside the trapezoid adjacent to the trapezoid are forbidden (see Fig. 4), i.e., it is sufficient to "insulate" the trapezoid by a single layer of forbidden edges. The only exception is the edges adjacent to the $(0, 0)$ and $(m, 0)$ vertex as those will be used in the construction (step 4).
3. Rotate the trapezoid by 45° counterclockwise. This isolated trapezoid can be embedded in a directed grid, and its starting and ending vertices are connected by a path if and only if the corresponding input word is valid.
4. Finally, we can lay multiple independent trapezoids side by side and connect them in parallel forming an OR$_t$ of DYCK$_{d,m}$ instances; see Fig. 5.

Fig. 3 Mapping of $\textsc{Dyck}_{d,m}$ variables to 2D- DConnectivity



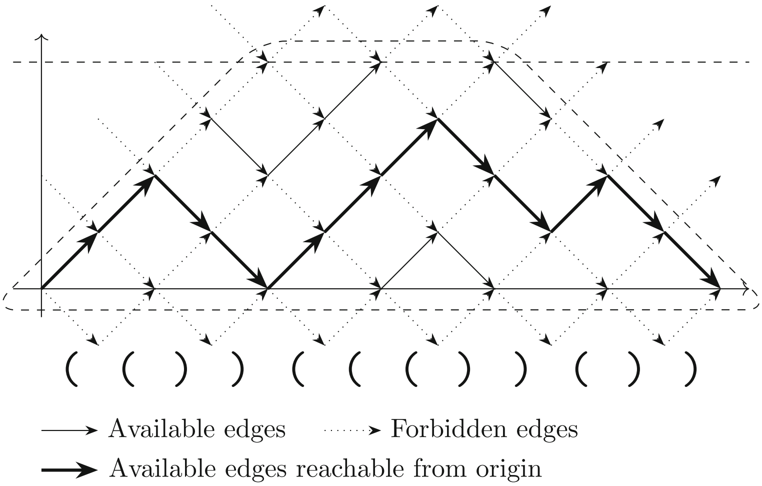


Fig. 4 Mapping of a complete input corresponding to Dyck word "(( )) ((( )) ( ))" to 2D- DConnectivity

Fig. 5 Folding of a long Dyck instance in an undirected grid

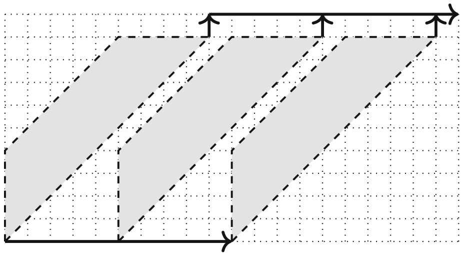
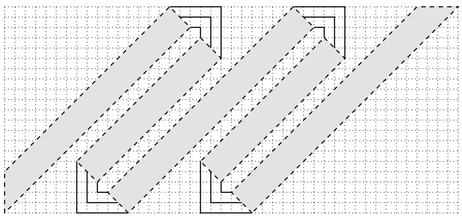


Fig. 6 Folding of a long Dyck instance in an undirected grid



This concludes the reduction $\textsc{Or}_t \circ \textsc{Dyck}_{d,m} \leq$ 2D- $\textsc{DConnectivity}_{n,k}$, where $n = (d+1)(t-1) + \frac{m}{2} + 1$ and $k = \frac{m}{2} + 1$. By the well-known composition result of Reichardt [21], we know that $Q(\textsc{Or}_t \circ \textsc{Dyck}_{d,m}) = \Theta(Q(\textsc{Or}_t) \cdot Q(\textsc{Dyck}_{d,m}))$. All that remains is to pick suitable t, d, and m for the proof to be complete. Let k be the vertical dimension of the grid and $k \leq n$. Then, we take $m = \Theta(k)$, $d = \log m$ and $t = \frac{n}{d}$. $\square$

Constructing a nontrivial quantum algorithm appears to be difficult, and we conjecture that the actual complexity may be $\Omega(nk)$, except for the case when k is small, compared to n. For very small k (up to $k = \Theta(\frac{\log n}{\log \log n})$), a better quantum algorithm is possible.

Theorem 9 $Q(\text{2D- }\textsc{DConnectivity}_{n,k}) = O\left(\sqrt{n}\log_2^{k/2} n\right)$. *Moreover, there is a time-efficient quantum query algorithm that solves* 2D- $\textsc{DConnectivity}_{n,k}$ *in time* $O\left(\sqrt{n}\log_2^{k/2+O(1)} n\right)$.

Proof We prove the claim by constructing a quantum algorithm for 2D- $\textsc{DConnectivity}_{n,k}$. The main idea is to construct an AND-OR formula for 2D- $\textsc{DConnectivity}_{n,k}$ and to use one of quantum algorithms for AND-OR formula evaluation. To achieve the optimal query complexity, we use the algorithm by Reichardt [26], which evaluates an AND-OR formula of size L with $O(\sqrt{L})$ queries. To achieve a time-efficient quantum algorithm, we can use quantum algorithms from [27] or [21] for which the number of queries is slightly larger ($O(\sqrt{Ld})$ for [27] and $O(\sqrt{L \log L})$ for [21]) and the number of nonquery steps is $O(\log^c L)$ per one query step. For the formula that we construct, $d = \log L$ and either of those quantum algorithms uses $O(\sqrt{L \log L})$ queries and $O(\sqrt{L}\log^c L)$ time steps.

We first deal with the case when $n = 2^m$ for some nonnegative integer m. The idea for the construction of the AND-OR formula is to split the grid in two: any path from $(0, 0)$ to (n, k) must pass through a vertex $(\frac{n}{2}, r)$ for some $r : 1 \leq r \leq k$. For the paths to and from $(\frac{n}{2}, r)$, we can apply this reasoning recursively. Let us denote by $F_{\mu,\kappa,i,j}$ our formula for the path from vertex (i, j) to $(i + 2^\mu, j + \kappa)$ and by $L_{\mu,\kappa}$ its size (the number of variable instances it has; it does not depend on i, j). Thus, we have the recurrent formulae

$$F_{\mu,\kappa,i,j} = \bigvee_{r=0}^{\kappa} \left(F_{\mu-1,r,i,j} \wedge F_{\mu-1,\kappa-r,i+2^{\mu-1},j+r}\right),$$

$$L_{\mu,\kappa} = \sum_{r=0}^{\kappa} \left(L_{\mu-1,r} + L_{\mu-1,\kappa-r}\right) = 2\sum_{r=0}^{\kappa} L_{\mu-1,r}.$$

For the base case $F_{0,\kappa,i,j}$ (i. e. for a $1 \times \kappa$ grid), we simply use an OR of all the paths (represented as an AND of all its edges). There are $\kappa + 1$ paths, each of length $\kappa + 1$, thus $L_{0,\kappa} = (\kappa + 1)^2$.

It follows by induction on μ that $L_{\mu,\kappa} < 2^{\mu+1} \cdot \binom{\kappa+\mu+2}{\kappa}$. For the induction basis, we have $L_{0,\kappa} < (\kappa+1)(\kappa+2) = 2\binom{\kappa+2}{\kappa}$, and for the induction step:

$$L_{\mu,\kappa} = 2\sum_{r=0}^{\kappa} L_{\mu-1,r} < 2^{\mu+1}\sum_{r=0}^{\kappa}\binom{r+\mu+1}{r} = 2^{\mu+1}\binom{\kappa+\mu+2}{\kappa}.$$

Using a well-known upper bound for binomial coefficients we obtain: $L_{m,k} < 2^{m+1}(e\cdot(k+m+2)/k)^k = O\left(n(e(1+\frac{\log_2 n}{k}))^k\right)$. There exists a quantum algorithm with $O(\sqrt{L})$ queries for a formula of size L [26], thus we obtain the complexity mentioned in the theorem statement.

For an arbitrary n we can find the smallest m for which $n \le 2^m$ and use the formula for the $2^m \times k$ grid obtained by adding ancillary edges from the vertex (n,k) to $(2^m,k)$ (using the edge variables of the added part of the grid as constants). Since the value of n thus increases no more than two times, the complexity estimation increases by at most a constant multiplier. □

Another case where we have an algorithm is when there is a specific restriction on the sparseness of the grid. Let a segment be a sequence of connected points in a line that cannot be extended. In other words, for $0 \le l < r \le n$, we have a segment between (i,l) and (i,r) iff (i,l) and (i,r) are connected, $(i,l-1)$ and (i,l) are not connected and (i,r) and $(i,r+1)$ are not connected. Let $\mathcal{S}_i$ be the number of segments in the line i.

For some $i \in \{0,\ldots,k-1\}, l, r \in \{0,\ldots,n\}$, let us call "a segment of vertical edges" a sequence of vertical edges from the edge between (i,l) and $(i+1,l)$ to the edge between (i,r) and $(i+1,r)$ such that all edges in the sequence exist, but there are no edges between $(i,l-1)$ and $(i+1,l-1)$ and between $(i,r+1)$ and $(i+1,r+1)$. Let $\mathcal{S}'_i$ be the number of "segments of vertical edges" between lines i and $i+1$. Let $\mathcal{S} = \sum_{i=1}^{k}\mathcal{S}_i + \sum_{i=0}^{k-1}\mathcal{S}'_i$.

We can define $\mathcal{S}^V$ in a similar way to $\mathcal{S}$ but by using horizontal edges. This is equivalent to rotating the grid by $\pi/2$.

Theorem 10 $Q(\text{2D- DCONNECTIVITY}_{n,k}) = O\left(\sqrt{nk\cdot\min(\mathcal{S},\mathcal{S}^V)}\log n\right)$.

Proof The algorithm performs a breadth-first search.

For $i \in \{0,\ldots,k\}$, let $\mathcal{A}_{2i}$ be the set of segments on the line i that can be reached from the point $(0,0)$. We can implement it as a self-balanced binary search tree, for example, it can be a Red–Black tree or an AVL tree. Such an implementation allows us to add segments to the set in $O(\log n)$ running time. Additionally, we can check whether a point (i,j) is in one of segments forming the set in $O(\log n)$ running time.

If (n,k) belongs to one of the segments from $\mathcal{A}_{2k}$, then it is reachable from $(0,0)$.

For $i \in \{0,\ldots,k-1\}$, let $\mathcal{A}_{2i+1}$ be the set of "segments of vertical edges" between lines i and $i+1$ such that all edges of a segment from the set can be achieved from $(0,0)$.

The set $\mathcal{A}_0$ contains only one segment that contains $(0,0)$. We can find the second border of the segment using the first one search algorithm of [24, 28, 29] with $O(\sqrt{r})$

expected query complexity, if the right border of the segment is $(0, r)$. The algorithm searches for the first r such that there is an edge between $(0, r-1)$ and $(0, r)$, but there is no edge between $(0, r)$ and $(0, r+1)$. We then add the segment $(0, 0)$, $(0, r)$ to the set $\mathcal{A}_0$.

For $i \in \{1, \ldots, k\}$, we show how to construct $\mathcal{A}_{2i}$ if we already constructed $\mathcal{A}_{2i-1}$.

Firstly, we search for l_1 the left border of the first segment. It is the minimal element such that the points (i, l_1) and (i, l_1+1) are connected and the edge between $(i-1, l_1)$ and (i, l_1) belongs to one of the "segments of vertical edges" from $\mathcal{A}_{2i-1}$. We do it with an expected $O(\sqrt{l_1} \log n)$ number of queries because of the complexity of the first one search algorithm and the complexity of checking the existence of an element in a self-balanced search tree. Then, we search for the minimal element r_1 such that $r_1 > l_1$, $(i, r_1 - 1)$ and (i, r_1) are connected, and (i, r_1) and $(i, r_1 + 1)$ are not connected. We can do it using the first one search algorithm with $O(\sqrt{r_1 - l_1})$ expected number of queries. Then, we add the segment between (i, l_1) and (i, r_1) to $\mathcal{A}_{2i}$.

For $j > 1$, if we have already found r_{j-1}, then we search for the minimal l_j such that $l_j > r_{j-1}$, the points (i, l_j) and $(i, l_j + 1)$ are connected and the edge between $(i-1, l_1)$ and (i, l_1) belongs to one of the "segments of vertical edges" from $\mathcal{A}_{2i-1}$. Then, we search for the minimal element r_j such that $r_j > l_j$, $(i, r_j - 1)$ and (i, j_1) are connected, and (i, r_j) and $(i, r_j + 1)$ are not connected. Then, we add the segment between (i, l_j) and (i, r_j) to $\mathcal{A}_{2i}$. The total complexity of this step is similar to the first step. It is $O(\sqrt{l_j - r_{j-1}} \log n + \sqrt{r_j - l_j})$.

Assume that $r_0 = 0$. Then, the total complexity of constructing $\mathcal{A}_{2i}$ is

$$O\left(\sum_{j=1}^{|\mathcal{A}_{2i}|} (\sqrt{l_j - r_{j-1}} \log n + \sqrt{r_j - l_j})\right)$$
$$\leq O\left(\sum_{j=1}^{|\mathcal{A}_{2i}|} (\sqrt{l_j - r_{j-1}} \log n + \sqrt{r_j - l_j} \log n)\right)$$
$$= O\left(\log n \sum_{j=1}^{|\mathcal{A}_{2i}|} (\sqrt{l_j - r_{j-1}} + \sqrt{r_j - l_j})\right)$$

According to Cauchy–Bunyakovsky–Schwarz inequality, we have

$$\leq O\left(\log n \sqrt{|\mathcal{A}_{2i}| \sum_{j=1}^{|\mathcal{A}_{2i}|} (l_j - r_{j-1} + r_j - l_j)}\right) \leq O\left(\log n \sqrt{|\mathcal{A}_{2i}| \cdot n}\right)$$
$$\leq O\left(\log n \sqrt{\mathcal{S}_i \cdot n}\right).$$

For $i \in \{0, \ldots, k-1\}$, we show how to construct $\mathcal{A}_{2i+1}$ if we have already constructed $\mathcal{A}_{2i}$ and $\mathcal{A}_{2i-1}$. Assume that $\mathcal{A}_{-1}$ is the empty set.

Firstly, we search for l_1 the left border of the first "segment of vertical edges". It is an element such that the points (i, l_1) and $(i+1, l_1)$ are connected and (i, l_1) belongs to one of segments from $\mathcal{A}_{2i}$ or the edge between $(i-1, l_1)$ and (i, l_1) belongs to one of "segments of vertical edges" from $\mathcal{A}_{2i-1}$. We do it with expected $O(\sqrt{l_1}\log n)$ number of queries by the argument similar to the previous step. Then, we search for the minimal element r_1 such that $r_1 > l_1$, and (i, r_1) belongs to one of segments from $\mathcal{A}_{2i}$ or the edge between $(i-1, r_1)$ and (i, r_1) belongs to one of "segments of vertical edges" from $\mathcal{A}_{2i-1}$. We can do it using the first one search algorithm with expected $O(\sqrt{r_1 - l_1}\log n)$ queries. Then, we add the segment between (i, l_1) and (i, r_1) to $\mathcal{A}_{2i-1}$.

For $j > 1$, if we have already found r_{j-1}, then we search for the minimal l_j such that

1. $l_j > r_{j-1}$,
2. (i, l_j) and $(i+1, l_j)$ are connected,
3. one or both conditions are true:
   - (i, l_j) belongs to one of segments from $\mathcal{A}_{2i}$ or
   - the edge between $(i-1, l_j)$ and (i, l_j) belongs to one of "segments of vertical edges" from $\mathcal{A}_{2i-1}$.
4. For $l_j - 1$ either condition 2 or condition 3 is wrong.

Then, we search for the element r_j similar to r_1 with respect to l_j. We can do it with $O(\log n(\sqrt{l_j - r_{j-1}} + \sqrt{r_j - l_j}))$ expected number of queries.

Similarly to the proof of the complexity of constructing $\mathcal{A}_{2i}$, we can show that the complexity of constructing $\mathcal{A}_{2i+1}$ is $O\left(\log n\sqrt{\mathcal{S}'_i \cdot n}\right)$.

The expected query complexity of the whole algorithm is:

$$
\begin{aligned}
&O\left(\sum_{i=0}^{k}(\log n\sqrt{\mathcal{S}_i \cdot n} + \log n\sqrt{\mathcal{S}'_i \cdot n})\right)\\
&= O\left(\sqrt{n}\log n\sum_{i=0}^{k}(\sqrt{\mathcal{S}_i} + \sqrt{\mathcal{S}'_i})\right) =\\
&= O\left(\sqrt{n}\log n\sqrt{k\sum_{i=0}^{k}(\mathcal{S}_i + \mathcal{S}'_i)}\right) \le O\left(\sqrt{n}\log n\sqrt{k\mathcal{S}}\right)\\
&= O\left(\sqrt{nk\mathcal{S}}\log n\right).
\end{aligned}
$$

We can invoke the same algorithm, but for the grid rotated by $\pi/2$. We invoke these algorithms in parallel and return the answer of the algorithm that reaches the last level first. The total expected query complexity is: $O\left(\sqrt{nk \cdot \min(\mathcal{S}, \mathcal{S}^V)}\log n\right)$.

Note that everywhere we use the Grover search algorithm (the first one search algorithm), we use it in the form that is presented in Lemma 1. □

Since $k \le \mathcal{S} \le nk$, the complexity can vary from $O(k\sqrt{n}\log n)$ to $O(kn\log n)$.

5.2 Lower bounds for 2D-CONNECTIVITY$_{n,k}$

Even though it is possible to use the construction from Sect. 5.1 to give a lower bound of $\Omega\left((\sqrt{n}k)^{1-\epsilon}\right)$ for the undirected case because the paths for each instance of DYCK never bifurcate or merge, this lower bound can be further improved to a nearly tight estimate.

Theorem 11 *For any $n \geq k$, $k = \Omega(\log n)$, $\epsilon > 0$, $Q(\text{2D- CONNECTIVITY}_{n,k}) = \Omega\left((nk)^{1-\epsilon}\right)$.*

Proof We start off by representing an input as a path in a trapezoid, see Fig. 4. But now instead of connecting multiple instances of DYCK in parallel, we will embed one long instance by folding it when it hits the boundary of the graph. To implement a fold, we will use simple gadgets depicted in Fig. 5.

This way a DYCK instance of length m and depth $\log m$ can be embedded in an $n \times k$ grid such that $\frac{nk}{\log m} = \Theta(m)$. Using Theorem 5, we conclude that solving 2D- CONNECTIVITY$_{n,k}$ requires at least $\Omega\left((nk)^{1-\epsilon}\right)$ quantum queries. □

5.3 Lower bounds for *d*-dimensional grids

For undirected d-dimensional grids, we give a tight bound on the number of queries required to solve connectivity.

Theorem 12 *For any $\epsilon > 0$, for undirected d-dimensional grids of size $n_1 \times n_2 \times \cdots \times n_d$ that are not "almost-one-dimensional", i.e., there exists $i \in [d]$ such that $\frac{\prod_{j=1}^{d} n_j}{n_i} = \Omega(\log n_i)$:*

$$Q(\text{dD- CONNECTIVITY}_{n_1,n_2,\ldots,n_d}) = \Omega((n_1 \cdot n_2 \cdot \ldots \cdot n_d)^{1-\epsilon}).$$

Proof For the purposes of this theorem, it is more convenient to refer to $n_1 \times \cdots \times n_d$ sized grids as $n'_1 \times \cdots \times n'_d$ sized where $n'_i = n_i + 1$. Then, the theorem follows from the 2D case by iteratively using the fact that a d-dimensional grid of size $n'_1 \times n'_2 \times \cdots \times n'_{d-1} \times n'_d$ contains as a subgraph a $(d-1)$-dimensional grid of size $n'_1 \times n'_2 \times \cdots \times n'_{d-2} \times n'_{d-1}n'_d$. One way to see this is to consider a bijective mapping of the vertices $(x_1, \ldots, x_{d-1}, x_d)$ to $(x_1, \ldots, x_{d-2}, x_d n'_{d-1} + x_{d-1})$ if x_d is even and to $(x_1, \ldots, x_{d-2}, x_d n'_{d-1} + n'_{d-1} - 1 - x_{d-1})$ if x_d is odd. It is a bijection because x_d and x_{d-1} can be recovered from $x_d n'_{d-1} + n'_{d-1} - 1 - x_{d-1}$ by computing the quotient and remainder on division by n'_{d-1}. One can view this procedure as "folding" where we take layers (vertices corresponding to some $x_d = l$) and fold them into the $(d-1)$-st dimension alternating the direction of the layers depending on the parity of the layer l. For this procedure to place the starting and ending vertices the furthest apart, it requires that n'_d is an odd number. Otherwise we embed a smaller subgraph $n'_1 \times \cdots \times n'_{d-1} \times (n'_d - 1)$ and add an edge $(n_1, \ldots, n_{d-1}, n_d - 1)$ to $(n_1, \ldots, n_{d-1}, n_d)$. In the end, we obtain a lower bound of $\Omega(((((((n'_d - 1)n'_{d-1} - 1)n'_{d-2} - 1)\cdots)n'_2 - 1)n'_1)^{1-\epsilon}) = \Omega((n_1 \cdot n_2 \cdot \ldots \cdot n_d)^{1-\epsilon})$. □

For directed d-dimensional grids, we can only slightly improve over the $n^{\frac{d}{2}}$ trivial lower bound.

Theorem 13 *For directed d-dimensional grids of size $n_1 \times n_2 \times \cdots \times n_d$ such that $n_1 \leq n_2 \leq \cdots \leq n_d$ and $\epsilon > 0$, $Q(\text{dD- DConnectivity}_{n_1,n_2,\ldots,n_d}) = \Omega((n_{d-1} \prod_{i=1}^{d} n_i)^{\frac{1}{2}-\epsilon})$.*

Corollary 14 *For directed d-dimensional grids of size $n \times n \times \cdots \times n$ and $\epsilon > 0$, $Q(\text{dD- DConnectivity}_{n,n,\ldots,n}) = \Omega(n^{\frac{d+1}{2}-\epsilon})$.*

Proof of Theorem 13 For each $I \in \{0, 1, \ldots, n_1\} \times \{0, 1, \ldots, n_1\} \times \cdots \times \{0, 1, \ldots, n_{d-2}\}$, we take a 2-dimensional hard instance G_I of $\text{2D- DConnectivity}_{n_{d-1},n_d}$ having query complexity $\Omega(n_{d-1}^{1-\epsilon} n_d^{\frac{1}{2}-\epsilon})$. We then connect them in parallel like so:

- Include the entire $(d-2)$-dimensional subgrid from $(0, \ldots, 0)$ to $(n_1, n_2, \ldots, n_{d-2}, 0, 0)$ and similarly the subgrid from $(0, 0, \ldots, 0, n_{d-1}, n_d)$ to $(n_1, n_2, \ldots, n_{d-2}, n_{d-1}, n_d)$;
- For each $I \in \{0, 1, \ldots, n_1\} \times \{0, 1, \ldots, n_1\} \times \cdots \times \{0, 1, \ldots, n_{d-2}\}$ embed the instance G_I in the subgrid $(I, 0, 0)$ to (I, n_{d-1}, n_d);
- Forbid all other edges.

This construction computes $\text{Or}_{\prod_{i=1}^{d-2}(n_i+1)} \circ \text{2D- DConnectivity}_{n_{d-1},n_d}$ whose complexity is at least $\Omega(\sqrt{\prod_{i=1}^{d-2}(n_i + 1)} n_{d-1}^{1-\epsilon} n_d^{\frac{1}{2}-\epsilon}) = \Omega((n_{d-1} \prod_{i=1}^{d} n_i)^{\frac{1}{2}-\epsilon})$. □

6 Directions for future works

Some directions for future work are:

1. *Better algorithm/lower bound for the directed 2D grid?* Can we find an $o(n^2)$ query quantum algorithm or improve our lower bound? A nontrivial quantum algorithm would be particularly interesting, as it may imply a quantum algorithm for edit distance.
2. *Quantum algorithms for directed connectivity?* More generally, can we come up with better quantum algorithms for directed connectivity? The span program method used by Belovs and Reichardt [30] for the undirected connectivity does not work in the directed case. As a result, the quantum algorithms for directed connectivity are typically based on Grover's search in various forms, from simply speeding up depth-first/breadth-first search to more sophisticated approaches [31]. Developing other methods for directed connectivity would be very interesting.
3. *Quantum speedups for dynamic programming* Dynamic programming is a widely used algorithmic method for classical algorithms, and it would be very interesting to speed it up quantumly. This has been the motivating question for both the connectivity problem on the directed 2D grid studied in this paper and a similar problem for the Boolean hypercube in [31] motivated by algorithms for Travelling Salesman Problem. There are many more dynamic programming algorithms and exploring their quantum speedups of them would be quite interesting.

Acknowledgements The research is supported by QuantERA ERA-NET Cofund in Quantum Technologies implemented within the European Union's Horizon 2020 Programme (QuantAlgo project) and ERDF Project 1.1.1.5/18/A/020 "Quantum algorithms: from complexity theory to experiment" Kamil Khadiev has been supported by the Kazan Federal University Strategic Academic Leadership Program ("PRIORITY-2030"); a part of his research (Theorem 10) was funded by the subsidy allocated to Kazan Federal University for the state assignment in the sphere of scientific activities, project No. 0671-2020-0065. Yixin Shen is supported by EPSRC grant EP/W02778X/1. We thank Frédéric Magniez for helpful discussions. Part of the work was done during Kamil Khadiev's visit to IRIF, Université Paris Cité.

Data availability Data sharing is not applicable to this article as no new data were created or analyzed in this study.

References

1. Ambainis, A., Balodis, K., Iraids, J., Khadiev, K., Kļevickis, V., Prūsis, K., Shen, Y., Smotrovs, J., Vihrovs, J.: Quantum lower and upper bounds for 2D-grid and Dyck language. In: 45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 170, pp. 8–1814 (2020)
2. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997). https://doi.org/10.1137/s0097539795293172
3. Cirac, J.I., Zoller, P.: Goals and opportunities in quantum simulation. Nat. Phys. **8**(4), 264–266 (2012). https://doi.org/10.1038/nphys2275
4. Georgescu, I.M., Ashhab, S., Nori, F.: Quantum simulation. Rev. Mod. Phys. **86**, 153–185 (2014). https://doi.org/10.1103/RevModPhys.86.153
5. Ambainis, A.: Understanding quantum algorithms via query complexity. In: Proceedings of the International Congress of Mathematicians, vol. 4, pp. 3283–3304 (2018)
6. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Automata theory, languages, and computation. Int. Ed. **24**(2), 171–183 (2006)
7. Aaronson, S., Grier, D., Schaeffer, L.: A quantum query complexity trichotomy for regular languages. Electron Colloquium Comput Complex (ECCC) **26**, 61 (2018)
8. Davis, M., Sigal, R., Weyuker, E.J.: Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science. Elsevier, Amsterdam (1994)
9. Khadiev, K., Kravchenko, D.: Quantum algorithm for Dyck language with multiple types of brackets. In: Unconventional Computation and Natural Computation, pp. 68–83 (2021)
10. Buhrman, H., Patro, S., Speelman, F.: The Quantum Strong Exponential-Time Hypothesis (2019)
11. Bennett, C.H., Bernstein, E., Brassard, G., Vazirani, U.: Strengths and weaknesses of quantum computing. SIAM J. Comput. **26**(5), 1510–1523 (1997)
12. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM (JACM) **21**(1), 168–173 (1974)
13. Backurs, A., Indyk, P.: Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In: Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, pp. 51–58. ACM (2015)
14. Boroujeni, M., Ehsani, S., Ghodsi, M., HajiAghayi, M., Seddighin, S.: Approximating edit distance in truly subquadratic time: quantum and MapReduce. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1170–1189. SIAM (2018)
15. Chakraborty, D., Das, D., Goldenberg, E., Koucký, M., Saks, M.E.: Approximating edit distance within constant factor in truly sub-quadratic time. In: 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS), Paris, France, Oct. 7–9, 2018, pp. 979–990 (2018)
16. Krinner, S., Lacroix, N., Remm, A., Paolo, A.D., Genois, E., Leroux, C., Hellings, C., Lazar, S., Swiadek, F., Herrmann, J., Norris, G.J., Andersen, C.K., Müller, M., Blais, A., Eichler, C., Wallraff, A.: Realizing repeated quantum error correction in a distance-three surface code. Nature **605**(7911), 669–674 (2022). https://doi.org/10.1038/s41586-022-04566-8
17. Collins, H., Nay, C.: IBM unveils 400 qubit-plus quantum processor and next-generation IBM quantum system two (2022). https://newsroom.ibm.com/2022-11-09-IBM-Unveils-400-Qubit-Plus-Quantum-Processor-and-Next-Generation-IBM-Quantum-System-Two

18. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition, 10th edn. Cambridge University Press, New York (2011)
19. Ablayev, F., Ablayev, M., Huang, J.Z., Khadiev, K., Salikhova, N., Wu, D.: On quantum methods for machine learning problems part I: Quantum tools. Big Data Min. Anal. **3**(1), 41–55 (2019)
20. Khadiev, K.: Lecture notes on quantum algorithms. arXiv:2212.14205 (2022)
21. Reichardt, B.W.: Reflections for quantum query algorithms. In: Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '11, pp. 560–569. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2011). http://dl.acm.org/citation.cfm?id=2133036.2133080
22. Kimmel, S.: Quantum adversary (upper) bound. In: International Colloquium on Automata, Languages, and Programming, pp. 557–568. Springer (2012)
23. Høyer, P., Mosca, M., de Wolf, R.: Quantum search on bounded-error inputs. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Automata, Languages and Programming, pp. 291–299. Springer, Berlin (2003)
24. Kapralov, R., Khadiev, K., Mokut, J., Shen, Y., Yagafarov, M.: Fast classical and quantum algorithms for online k-server problem on trees. arXiv:2008.00270 (2020)
25. Ambainis, A.: Quantum lower bounds by quantum arguments. J. Comput. Syst. Sci. **64**(4), 750–767 (2002)
26. Reichardt, B.W.: Span programs are equivalent to quantum query algorithms. SIAM J. Comput. **43**(3), 1206–1219 (2014). https://doi.org/10.1137/100792640
27. Ambainis, A., Childs, A.M., Reichardt, B., Spalek, R., Zhang, S.: Any AND-OR formula of size N can be evaluated in time $n^{1/2+o(1)}$ on a quantum computer. SIAM J. Comput. **39**(6), 2513–2530 (2010). https://doi.org/10.1137/080712167
28. Lin, C.Y.-Y., Lin, H.-H.: Upper bounds on quantum query complexity inspired by the Elitzur–Vaidman bomb tester. Theory Comput. **12**(18), 1–35 (2016)
29. Kothari, R.: An optimal quantum algorithm for the oracle identification problem. In: 31st International Symposium on Theoretical Aspects of Computer Science, p. 482 (2014)
30. Belovs, A., Reichardt, B.W.: Span programs and quantum algorithms for ST-connectivity and claw detection. In: Algorithms-ESA 2012-20th Annual European Symposium, Ljubljana, Slovenia, September 10–12, 2012. Proceedings, pp. 193–204 (2012). https://doi.org/10.1007/978-3-642-33090-2_18
31. Ambainis, A., Balodis, K., Iraids, J., Kokainis, M., Prusis, K., Vihrovs, J.: Quantum speedups for exponential-time dynamic programming algorithms. In: Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6–9, 2019, pp. 1783–1793 (2019). https://doi.org/10.1137/1.9781611975482.107

Authors and Affiliations

Andris Ambainis[1] · Kaspars Balodis[1] · Jānis Iraids[1] · Kamil Khadiev[2] · Vladislavs Kļevickis[1] · Krišjānis Prūsis[1] · Yixin Shen[3] · Juris Smotrovs[1] · Jevgēnijs Vihrovs[1]

Andris Ambainis
andris.ambainis@lu.lv

Kaspars Balodis
kaspars.balodis2@lu.lv

Jānis Iraids
krisjanis.prusis@lu.lv

Vladislavs Kļevickis
vladklevitsky@gmail.com

Krišjānis Prūsis
krisjanis.prusis@gmail.com

Juris Smotrovs
juris.smotrovs@lu.lv

Jevgēnijs Vihrovs
jevgenijs.vihrovs@lu.lv

[1] Center for Quantum Computer Science, Faculty of Computing, University of Latvia, Riga, Latvia

[2] Institute of Computational Mathematics and Information Technologies, Kazan Federal University, Kazan, Russia

[3] Information Security Group, Royal Holloway, University of London, Egham, UK