Quantum Lower and Upper Bounds for 2D-Grid and Dyck Language

Andris Ambainis, Kaspars Balodis, Jānis Iraids
Center for Quantum Computer Science, Faculty of Computing, University of Latvia

Kamil Khadiev
Kazan Federal University, Kazan, Russia

Vladislavs Klevickis, Krišjānis Prūsis
Center for Quantum Computer Science, Faculty of Computing, University of Latvia

Yixin Shen
Université de Paris, CNRS, IRIF, F-75006 Paris, France

Juris Smotrovs, Jevgēnijs Vihrovs
Center for Quantum Computer Science, Faculty of Computing, University of Latvia

Abstract
We study the quantum query complexity of two problems. First, we consider the problem of determining if a sequence of parentheses is a properly balanced one (a Dyck word), with a depth of at most $k$. We call this the $\text{Dyck}_{k,n}$ problem. We prove a lower bound of $\Omega(e^k \sqrt{n})$, 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. [1]. Their proof does not give rise to a general algorithm. When $k$ is not a constant, $\text{Dyck}_{k,n}$ is not context-free. We give an algorithm with $O(\sqrt{n}(\log n)^{0.5k})$ quantum queries for $\text{Dyck}_{k,n}$ 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. 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.

2012 ACM Subject Classification Theory of computation → Quantum query complexity

Keywords and phrases Quantum query complexity, Quantum algorithms, Dyck language, Grid path

Funding 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”. The research was funded by the subsidy allocated to Kazan Federal University for the state assignment in the sphere of scientific activities.

1 Introduction
We study the quantum query complexity of two problems:

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 the quantum case, the most commonly used model for studying the complexity of various problems is the query model. For this setting, Aaronson, Grier and Schaeffer [1] 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; $\Theta(\sqrt{n})$ 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 [1], 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 the
languages which have regular expressions not containing the Kleene star (if it is allowed to
use the complement operation). Star-free languages are one of the most commonly studied
subclasses of regular languages and there are many equivalent characterizations of them.
One of the star-free languages mentioned in [1] is the Dyck language (with one type of
parenthesis) with a constant bounded height. The Dyck language is the set of balanced
strings of parentheses ( and ). The language is a fundamental example of a context-free
language that is not regular. If at no point the number of opening parentheses exceeds the
number of closing parentheses by more than $k$, we denote the problem of determining if an
input of length $n$ belongs to this language by $\text{DYCK}_{k,n}$. 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.

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 $\Omega(n^{1-c})$.

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. [7].

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

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 by another, removing a symbol or inserting a new symbol)
with which one can transform $x$ to $y$. If $|x| \leq n, |y| \leq k$, the edit distance is solvable in
time $O(nk)$ by dynamic programming [16]. If $n = k$ then, under the strong exponential time
hypothesis ($\text{SETH}$), there is no classical algorithm computing edit distance in time $O(n^{2-\epsilon})$
for $\epsilon > 0$ [4] 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 $\text{SAT}$ that is substantially better than naive search.
Quantumly, a 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 a substantial interest from various researchers. Boroujeni et al. [6] invented a better-than-classical quantum algorithm for approximating the edit distance which was later superseded by a better classical algorithm of [8]. However, there has 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:

1. 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 $(n,k)$.

2. 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 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.

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 $\Omega((\sqrt{n}k)^{1-\epsilon})$.

The trivial upper bound is $O(nk)$ queries, since there are $O(nk)$ variables. There is no nontrivial quantum algorithm, except for the case when $k$ is very small. Then, the connectivity problem can be solved with $O(\sqrt{n}\log^k n)$ quantum queries [11] but this bound becomes trivial already for $k = O\left(\frac{\log n}{\log \log n}\right)$.

For the undirected 2D grid, we show a lower bound of $\Omega((nk)^{1-\epsilon})$, whenever $k \geq \log n$.

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 $\Omega((n_1n_2\ldots n_d)^{1-\epsilon})$ for an undirected $n_1 \times n_2 \times \ldots \times n_d$ grid in $d$ dimensions and a lower bound of $\Omega(n^{(d+1)/2-\epsilon})$ for a directed $n \times n \times \ldots \times n$ grid in $d$ dimensions.

In a recent work, an $\Omega(n^{1.5})$ lower bound for edit distance was shown by Buhrman et al. [7], 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 follow directly from one another, as different shortest path problems are used.

2 Definitions

For a word $x \in \Sigma^*$ and a symbol $a \in \Sigma$, let $|x|_a$ be the number of occurrences of $a$ in $x$.

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 $(g \circ h)(x) = g(h(x_1, \ldots, x_m), \ldots, h(x_{(n-1)m+1}, \ldots, x_{nm}))$. 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 $f^1 = f$.

\footnote{Aaronson et al. [1] 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 $k$.}
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 an oracle access to the input \( x \) — it is realized by a specific unitary transformation usually defined as \( |i\rangle|z\rangle\langle w| \to |i\rangle|z + x_i (\mod 2)|\langle w| \) where the \( |i\rangle \) register indicates the index of the variable we are querying, \( |z\rangle \) is the output register, and \( |w\rangle \) is some auxiliary work-space. An algorithm in the query model consists of alternating applications of arbitrary unitaries independent of the input and the query unitary, and a 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 \( Q(f) \).

Let a symmetric matrix \( \Gamma \) be called an adversary matrix for \( f \) if the rows and columns of \( \Gamma \) 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 \( \Gamma^{(i)}_{xy} = \Gamma_{xy} \) if \( x_i \neq y_i \) and \( \Gamma^{(i)}_{xy} = 0 \) otherwise. Then let \( \text{Adv}^+ (f) = \max_{\Gamma} \frac{\|\Gamma\|}{\Gamma_{x'y'} - \Gamma^{(n)}_{x'y'}}, \Gamma \text{ - an adversary matrix for } f \) be called the adversary bound and let \( \text{Adv}(f) = \max_{\Gamma} \frac{\|\Gamma\|}{\Gamma_{x'y'} - \Gamma^{(n)}_{x'y'}}, \Gamma \text{ - nonnegative } \) be the following: \( \text{Adv}(f) \leq \text{Adv}^+ (f); \ Q(f) = \Theta(\text{Adv}^+ (f)) \) \[14\]; \text{Adv}^+ composes exactly even for partial Boolean functions \( f \) and \( g \), meaning, \( \text{Adv}^+ (f \circ g) = \text{Adv}^+ (f) \cdot \text{Adv}^+ (g) \) \[10, Lemma 6\]

Reductions. We will say that a Boolean function \( f \) is reducible to \( g \) and denote by \( f \leq g \) if there exists an algorithm that given an oracle \( O_g \) for an input of \( f \) transforms it into an oracle \( O_y \) for \( g \) using at most \( O(1) \) calls of oracle \( O_g \) 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 \( y \).

Dyck languages of bounded depth. Let \( \Sigma \) 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 \( \text{Dyck}_{k,n}(x_1,\ldots,x_n) \) where \( x_1,\ldots,x_n \in \{0,1\} \) describe a word of length \( n \) in the natural way: the \( i^{\text{th}} \) symbol of \( x \) is \( ( \) if \( x_i = 0 \) and \( ) \) if \( x_i = 1 \). \( \text{Dyck}_{k,n}(x) = 1 \) 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 ±\( k \)-substring is a substring whose balance is equal to \( k \) in absolute value. For \( 0 \leq i \leq j \leq n - 1 \), 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 \( h^-(x) = \min_{0 \leq i \leq n - 1} f(x[0,i]) \). A substring \( x[i,j] \) is minimal if it does not contain a subsequence \( x[i',j'] \) such that \( (i,j) \neq (i',j') \), and \( f(x[i',j']) = f(x[i,j]) \).

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 \( (i,j), i \in [n], j \in [k] \) 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 \( \text{Directed-2D-Connectivity} \) in which one has to determine if \( G \) contains a path from \( (0,0) \) to \( (n,k) \): \( \text{Directed-2D-Connectivity}_{n,k}(x_1,\ldots,x_m) = 1 \) (where \( m \) is the number of edges in \( G_{n,k} \)) iff such a path exists.

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 \( \text{Undirected-2D-Connectivity}_{n,k}(x_1,\ldots,x_m) = 1 \) iff \( G \) contains a path from \( (0,0) \) to \( (n,k) \). We also consider \( d \) dimensional versions of these two problems, on \( n \times n \times \ldots \times n \) grids (with the grid of the same length in all the dimensions). In the directed version
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) ≠ 0.

3.1 ±k-Substring Search algorithm

The goal of this section is to describe a quantum algorithm which searches for a substring x[i, j] that has a balance f(x[i, j]) ∈ {±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 ±k-substrings x[i, j] and x[k, l]: i < k ⇒ j < l. This induces a natural linear order among all ±k-substrings according to their starting (or, equivalently, ending) positions.

Minimal +k-substrings do not intersect with minimal −k-substrings.

If x[l1, r1] and x[l2, r2] with l1 < l2 are two consecutive minimal (k − 1)-substrings and their signs are the same, then x[l1, r2] is a k-substring with this sign.

This algorithm is the basis of our algorithms for Dyck_{k,n}. The algorithm works in a recursive way. It searches for two consecutive minimal ±(k − 1)-substrings x[l1, r1] and x[l2, r2] such that they either overlap or there are no ±(k − 1)-substrings between them. If both substrings x[l1, r1] and x[l2, r2] 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.

We first discuss three building blocks for our algorithm. The first one is FindAtLeftmost_k(l, r, t, d, s) and accepts as inputs: the borders l and r, where l and r are integers such that 0 ≤ l ≤ r ≤ n − 1; a position t ∈ {l, ..., r}; a maximal length d for the substring, where d is an integer such that 0 < d ≤ r − l + 1; the sign of the balance s ∈ {−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 t ∈ [i, j], j − i + 1 ≤ d, f(x[i, j]) ∈ {±k, −k} and σ = sign(f(x[i, j])) ∈ 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 ∈ [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 FindAtRightmost_k. It is similar to FindAtLeftmost_k, but finds the rightmost ±k-substring, i.e. there is no other minimal x[i’, j’] such that j’ > j, l ∈ [i’, j’], f(x[i’, j’]) = f(x[i, j])

The third one is FindFirst_k(l, r, direction) and accepts as inputs: the borders l and r, where l and r are integers such that 0 ≤ l ≤ r ≤ n − 1; the sign of the balance s ∈ {−1, +1}, a direction ∈ {left, right}. When the direction is right (respectively left), FindFirst_k finds the first ±k-substring from the left to the right (respectively from the right to the left) in [l, r] of sign s.

These three building blocks are interdependent since FindAtLeftmost_k uses FindFirst_k−1 and FindAtRightmost_k−1 as subroutines, FindFirst_k uses FindAtLeftmost_k and
Quantum Lower and Upper Bounds for 2D-Grid and Dyck Language

FINDATRIGHTMOST_k as subroutines. A description of FINDATLEFTMOST_k(l, r, t, d, s) follows. The algorithm is presented in Appendix A. The description of FINDATRIGHTMOST_k(l, r, t, d, s) is similar and is omitted.

When k = 2, the procedure FINDATLEFTMOST_2(l, r, t, d, s) checks that x_i = x_{i-1} and sign(f(x[t − 1, l])) ∈ s. If yes, it has found the substring. Otherwise, it checks if x_i = x_{i+1} and sign(f(x[t, t + 1])) ∈ 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 ±(k − 1)-substring of length at most d − 1, i.e.

v = (i, j, σ) ← FINDATLEFTMOST_{k−1}(l, r, t, d−1, {+1, −1}). If v = NULL, then (i_1, j_1, σ_1) ← (i, j, σ) and the algorithm goes to Step 2. Otherwise, the algorithm goes to Step 6.

Step 2. Check whether i_1 − 1 is inside a ±(k − 1)-substring of length at most d − 1 and choose the rightmost one: v = (i, j, σ) ← FINDATRIGHTMOST_{k−1}(l, r, i_1−1, d−1, {+1, −1}).

If v = NULL, then the algorithm goes to Step 3. If v = NULL and σ = σ_1, then (i_2, j_2, σ_2) ← (i, j, σ) and go to Step 8. Otherwise, go to Step 4.

Step 3. Search for the first ±(k − 1)-substring on the left from i_1 − 1 at distance at most d, i.e. v = (i, j, σ) ← FINDFIRST_{k−1}(min(l, j_1−d+1), i_1−1), {+1, −1}, left). If v = NULL and σ = σ_1 then (i_2, j_2, σ_2) ← (i, j, σ) and go to Step 8. Otherwise, go to Step 4.

Step 4. Check whether j_1 + 1 is inside a ±(k − 1)-substring of length at most d − 1, i.e.

v = (i, j, σ) ← FINDATLEFTMOST_{k−1}(l, r, j_1+1, d−1, {+1, −1}).

If v = NULL, then (i_2, j_2, σ_2) ← (i, j, σ) and go to Step 8. Otherwise, go to Step 5.

Step 5. Search for the first ±(k − 1)-substring on the right from j_1 + 1 at distance at most d, i.e. v = (i, j, σ) ← FINDFIRST_{k−1}(j_1+1, min(i_1+d−1, r), {+1, −1}, right).

If v = NULL, then (i_2, j_2, σ_2) ← (i, j, σ), then go to Step 8. Otherwise, return NULL.

Step 6. Search for the first ±(k − 1)-substring on the right at distance at most d from t, i.e. v = (i, j, σ) ← FINDFIRST_{k−1}(t, min(t + d − 1, r), {+1, −1}, right).

If v = NULL, then (i_1, j_1, σ_1) ← (i, j, σ) and go to Step 7. Otherwise, returns NULL.

Step 7. Search for the first ±(k − 1)-substring on the left from t at distance at most d, i.e. v = (i, j, σ) ← FINDFIRST_{k−1}(max(l, t − d + 1), t), {+1, −1}, left).

If v = NULL, then (i_2, j_2, σ_2) ← (i, j, σ) and go to Step 8. Otherwise, returns NULL.

Step 8. If σ_1 = σ_2, σ_1 ∈ s and max(j_1, j_2)−min(i_1, i_2)+1 ≤ d, output (min(i_1, i_2), max(j_1, j_2), σ_1), otherwise return NULL.

By construction and induction on k, the two ±(k − 1)-substrings x[i_1, j_1] and x[i_2, j_2] (if they exist) involved in the procedure FINDATLEFTMOST_k are always consecutive and minimal. FINDATLEFTMOST_k thus returns a ±k-substring, if both substrings have the same sign.

Using this basic procedure, we then search for a ±k-substring by searching for a t and d such that 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 [9]. 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 to stop the Grover iteration early and make \(O(\sqrt{(r−l)/d})\) queries in all cases.
Lemma 1 (Modified from [9], Appendix B). 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.

The algorithm that uses above ideas is presented in Algorithm 1.

Algorithm 1 $\text{FindFixedLen}_k(l, r, d, s)$. Search for any $\pm k$-substring of length $\in [d/2, d]$

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

We can then write an algorithm $\text{FindAny}_k(l, r, s)$ 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. $d \in \{2^{\lceil \log_2 k \rceil}, 2^{\lceil \log_2 k \rceil+1}, \ldots, 2^{\lceil \log_2 (r-l) \rceil}\}$. For the chosen $d$, we run Algorithm 1. 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 2 $\text{FindAny}_k(l, r, s)$. Search for any $\pm k$-substring.

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

Finally, we present the algorithm that finds the first $\pm k$-substring – $\text{FindFirst}_k$. 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 $mid = \lfloor (l_{Border} + r_{Border})/2 \rfloor$ be the middle of the search segment $[l_{Border}, r_{Border}]$. There are three cases:

1. There is a $k$-substring in $[l_{Border}, mid]$, then the leftmost $k$-substring is in this segment.
2. There are no $k$-substrings in $[l_{Border}, mid]$, but $mid$ is inside a $k$-substring. Then the leftmost $k$-substring that contains $mid$ is the required substring.
3. There are no $k$-substrings in $[l_{Border}, mid]$ and $mid$ is not inside a $k$-substring. Then the required substring is in $[mid + 1, r_{Border}]$.

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 $\text{FindAtLeftmost}_k$ by $\text{FindAtRightmost}_k$ that finds the rightmost $\pm k$-substring that contains $mid$. A detailed description of this algorithm is presented in Appendix C.

Proposition 2. For any $\varepsilon > 0$ and $k$, algorithms $\text{FindAtLeftmost}_k$, $\text{FindFixedLen}_k$, $\text{FindAny}_k$ and $\text{FindFirst}_k$ have two-sided error probability $\varepsilon < 0.5$ and return, when correct:

1. If $t$ is inside a $\pm k$-substring of sign $s$ of length at most $d$ in $x[l, r]$, then $\text{FindAtLeftmost}_k$ 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 simplicity of FindAtLeftmost_2 and FindAtRightmost_2 procedures. We first prove the correctness of all the algorithms, assuming there are no errors. At 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 \( i_1 \) and \( i_2 \) such that \( i < i_1 < i_2 < j \), \( |f(x[i, i_1])| = |f(x[i_2, j])| = k-1 \) and sign\( (f(x[i, i_1])) = sign(f(x[i_2, j])) \) \( \in \mathbb{S} \). If \( t \in \{i_2, \ldots, j\} \), then the algorithm finds \( x[i_2, j] \) in Step 1 and the first invocation of FindFirst_{k-1} in Step 3 finds \( x[i, i_1] \). If \( t \in \{i, \ldots, i_1\} \), then the algorithm finds \( x[i, i_1] \) 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 forth 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 sign\( (f(x[i, j_1])) = sign(f(x[i_2, j])) \) \( \in \mathbb{S} \). If \( t \in \{i, \ldots, j_1\} \), then the algorithm finds \( x[i, j_1] \) in Step 1. After that, it finds \( x[i_2, j] \) in Step 4. If \( t \in \{j_1+1, \ldots, j\} \), 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 \( O\left(\frac{\sqrt{d}}{\sqrt{d/2}}\right) \) 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: See Appendix C.

We now turn to error analysis. The case of 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 \( \varepsilon \).

The analysis of FindFixedLen_k follows directly from [9] and Lemma 1: since FindAtLeftmost_k has two-sided error \( \varepsilon \), there exists a search algorithm with two-sided error \( \varepsilon \).
3.2 The Algorithm for Dyck\(_{k,n}\)

To solve Dyck\(_{k,n}\), we modify the input \(x\). As the new input we use \(x' = 1^k x 0^k\). Dyck\(_{k,n}(x) = 1\) iff there are no \(\pm (k + 1)\)-substrings in \(x'\). This idea is presented in Algorithm 3.

**Algorithm 3** Dyck\(_{k,n}\). The Quantum Algorithm for Dyck\(_{k,n}\).

\[
x \leftarrow 1^k x 0^k \\
v = \text{FindAny}_{(k+1)}(0, n + 2k - 1, \{+1, -1\})
\]

return \(v = = \text{NULL}\)

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

4 Lower bounds for Dyck languages

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

**Proof.** We will use the partial Boolean function \(\text{Ex}^{m|b}_{\cdot m}\) = \(\begin{cases} 1, & \text{if } |x|_0 = a \\ 0, & \text{if } |x|_0 = b \end{cases}\)

We prove the theorem by a reduction \((\text{Ex}^{m|m+1}_{\cdot 2m})^\ell \leq \text{Dyck}_{c_1 \ell m, c_2 (2m)}^\ell\), with the reduction described in appendix E. It is known that \(\text{Adv}^\ell(\text{Ex}^{m|m+1}_{\cdot 2m}) \geq \text{Adv}(\text{Ex}^{m|m+1}_{\cdot 2m}) > m\) [2, Theorem 5.4]. The Adversary bound composes even for partial Boolean functions [10, Lemma 1], therefore \(Q(\text{Ex}^{m|m+1}_{\cdot 2m})^\ell) = \Omega(m^\ell)\). Via the reduction the same bound applies to Dyck\(_{c_1 \ell m, c_2 (2m)}^\ell\).

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

**Proof.** For any \(\varepsilon > 0\), there exists an \(m\) such that \(\text{Adv}^\ell(\text{Ex}^{m|m+1}_{\cdot 2m}) \geq (2m)^{1-\varepsilon}\). Without loss of generality we may assume that \((2m)^\ell = n\). From Theorem 4 with \(\ell = \log_2 2m n\) we obtain \(c_2(2m)^\ell = c_2 n\) and height \(c_1 m \ell = \Theta(\log n)\). The query complexity is at least \((2m)^{1-\varepsilon}\) = \((2m)^{1-\varepsilon} = n^{1-\varepsilon}\). Therefore \(Q(\text{Dyck}_{c \log n, n}) = \Omega(n^{1-\varepsilon})\).

For constant depths the following bound can be derived:

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

**Proof.** Let \(m = 4\) in the Theorem 4. Then, \(Q(\text{Dyck}_{c_1 \ell, c_2 8^\ell}) = \Omega(4^\ell)\) for some constants \(c_1, c_2 > 0\). Consider the function \(\text{AND} = \frac{n}{c_2 8^\ell} \circ \text{Dyck}_{c_1 \ell, c_2 8^\ell}\) with a promise that \(\text{AND}_k\) has as an input either \(k\) or \(k-1\) ones. The query complexity of this function is \(\Omega\left(\sqrt{\frac{n}{c_2 8^\ell} 4^\ell}\right) = \Omega(2^\ell \sqrt{n})\).

The computation of the composition \(\text{AND} = \frac{n}{c_2 8^\ell} \circ \text{Dyck}_{c_1 \ell, c_2 8^\ell}\) can be straightforwardly reduced to Dyck\(_{c_1 \ell, n}\) by a simple concatenation of Dyck\(_{c_1 \ell, c_2 8^\ell}\) instances.

5 Quantum complexity of st-Connectivity in grids

5.1 Quantum complexity of Directed-2D-Connectivity\(_{n,k}\)

**Theorem 7.** For any \(n \geq k\) and \(\epsilon > 0\), \(Q(\text{Directed-2D-Connectivity}_{n,k}) = \Omega((\sqrt{n k})^{1-\epsilon})\).
In particular, if we have a square grid then

\[ Q\left( \text{Directed-2D-Connectivity}_{n,n} \right) = \Omega(n^{1.5-\epsilon}). \]

**Corollary 1.** For any \( \epsilon > 0 \), \( Q(\text{Directed-2D-Connectivity}_{n,n}) = \Omega(n^{1.5-\epsilon}). \)

**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 Figure 1. We can connect neighboring points by vectors \((1, 1)\) and \((1, -1)\) corresponding to 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 \( \text{Dyck}_{d,m} \) to the \( \text{Directed-2D-Connectivity}_{n,k} \) problem:

1. An opening parenthesis in position \( i \) corresponds to a “column” of upwards sloping available edges \((i-1, l) \rightarrow (i, l+1)\) 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 \((i-1, l) \rightarrow (i, l-1)\) for all \( l \in \{1, \ldots, d\}\) such that \( i-1 + l \) is even. See Figure 2.

2. The edges outside the trapezoid adjacent to the trapezoid are forbidden (see Figure 3), i.e., it is sufficient to “insulate” the trapezoid by a single layer of forbidden edges. The only exception are 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 degrees 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 \( \text{OR}_t \) of \( \text{Dyck}_{d,m} \) instances; see Figure 4.

This concludes the reduction \( \text{OR}_t \circ \text{Dyck}_{d,m} \leq \text{Directed-2D-Connectivity}_{n,k} \), where \( n = (d+1)(l-1) + \frac{m}{2} + 1 \) and \( k = \frac{m}{2} + 1 \). By the well known composition result of Reichardt [14] we know that \( Q(\text{OR}_t \circ \text{Dyck}_{d,m}) = \Theta(Q(\text{OR}_t) \cdot Q(\text{Dyck}_{d,m})) \). 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 \( t = \frac{m}{2} \).

Constructing a non-trivial 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.

\textbf{Theorem 8} (Appendix F). $Q(\text{DIRECTED-2D-CONNECTIVITY}_{n,k}) = O\left(\sqrt{nk} \left(1 + \frac{\log n}{k}\right)^k\right)$.

\section{Lower bounds for Undirected-2D-Connectivity $n,k$}

Even though it is possible to use the construction from Section 5.1 to give a lower bound of $\Omega(\sqrt{nk})$ for the undirected case because the paths for each instance of $\text{Dyck}$ never bifurcate or merge, this lower bound can be further improved to a nearly tight estimate.

\textbf{Theorem 9.} For any $n \geq k$, $k = \Omega(\log n)$, $\epsilon > 0$, $Q(\text{UNDIRECTED-2D-CONNECTIVITY}_{n,k}) = \Omega((nk)^{1-\epsilon})$.

\textbf{Proof.} We start off by representing an input as a path in a trapezoid, see Figure 3. But now instead of connecting multiple instances of $\text{Dyck}$ 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 Figure 5.

This way a $\text{Dyck}$ 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 $\text{UNDIRECTED-2D-CONNECTIVITY}_{n,k}$ requires at least $\Omega((nk)^{1-\epsilon})$ quantum queries.

\section{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.

\textbf{Theorem 10.} For any $\epsilon > 0$, for undirected $d$-dimensional grids of size $n_1 \times n_2 \times \ldots \times n_d$ that are not “almost-one-dimensional”, i.e., there exists $i \in [d]$ such that $\prod_{j=1}^{d} \frac{n_j}{n_i} = \Omega(\log n_i)$:

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

\textbf{Proof.} This follows from the 2D case by using the fact that a $d$-dimensional grid of size $n_1 \times n_2 \times \ldots \times n_{d-1} \times n_d$ contains as a subgraph a $(d-1)$-dimensional grid of size $n_1 \times n_2 \times \ldots \times n_{d-2} \times n_{d-1} n_d$. One way to see this is to consider a bijective mapping of the vertices

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{figure3.png}
  \caption{Mapping of a complete input corresponding to Dyck word “((()((()))))” to $\text{DIRECTED-2D-CONNECTIVITY}$}
\end{figure}
\((x_1, \ldots, x_{d-1}, x_d)\) 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 directed \(d\)-dimensional grids we can only slightly improve over the \(n^{\frac{d}{2}}\) trivial lower bound.

**Theorem 11.** For directed \(d\)-dimensional grids of size \(n_1 \times n_2 \times \cdots \times n_d\) such that \(n_1 \leq n_2 \leq \ldots \leq n_d\) and \(\epsilon > 0\), \(Q(\text{Directed-}d\text{-D-Connectivity}_{n_1, n_2, \ldots, n_d}) = \Omega((n_{d-1} \prod_{i=1}^{d} n_i)^{\frac{1}{d}} - \epsilon)\).

**Corollary 2.** For directed \(d\)-dimensional grids of size \(n \times n \times \cdots \times n\) and \(\epsilon > 0\),
\[Q(\text{Directed-}d\text{-D-Connectivity}_{n, n, \ldots, n}) = \Omega(n^{\frac{d}{d+1}} - \epsilon).\]

**Proof of Theorem 11.** For each \(I \in [n_1] \times [n_2] \times \cdots \times [n_{d-2}]\) we take a 2-dimensional hard instance \(G_I\) of \(\text{Directed-}d\text{-D-Connectivity}_{n_{d-1}, n_d}\) having query complexity \(\Omega(n_{d-1}^{\frac{1}{d}} n_d^{\frac{1}{d}} - \epsilon)\).

We then connect them in parallel like so:
- Make available the entire \((d-2)\)-dimensional subgrid from \((1, 1, \ldots, 1, 1)\) to \((n_1, n_2, \ldots, n_{d-2}, 1, 1)\) and similarly the subgrid from \((1, 1, \ldots, 1, n_{d-1}, n_d)\) to \((n_1, n_2, \ldots, n_{d-2}, n_{d-1}, n_d)\);
- For each \(I \in [n_1] \times [n_2] \times \cdots \times [n_{d-2}]\) embed the instance \(G_I\) in the subgrid \((I, 1, 1)\) to \((I, n_{d-1}, n_d)\);
- Forbid all other edges.

This construction computes \(\text{OR}_{I \leq n_1 \prod_{i=1}^{d-2} n_i} \circ \text{Directed-}d\text{-D-Connectivity}_{n_{d-1}, n_d}\) whose complexity is at least \(\Omega(\sqrt{\prod_{i=1}^{d-2} n_i n_{d-1}^{\frac{1}{d}} n_d^{\frac{1}{d}} - \epsilon}) = \Omega((n_{d-1} \prod_{i=1}^{d} n_i)^{\frac{1}{d}} - \epsilon)\).

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 [5] 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 [3]. 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 [3] motivated by algorithms for Travelling Salesman Problem. There are many more dynamic programming algorithms and exploring quantum speedups of them would be quite interesting.

References

11 Vladislavs Klevickis. Čaulu programmas cela atrašanai graf¯ a (span programs for finding a path in a graph). Undergraduate 3rd year project, University of Latvia, 2017.


A. Ambainis et al.

A. An Algorithm for the FindAtLeftmost\(k\) Subroutine

Algorithm 4 FindAtLeftmost\(k\)(l, r, t, d, s).

\[v = (i_1, j_1, \sigma_1) \leftarrow \text{FindAtLeftmost}_{k-1}(l, r, t, d - 1, \{+1, -1\})\]

if \(v \neq \text{NULL}\) then \(\triangleright\) if \(t\) is inside a \((k-1)\)-substring

\[v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindAtRightmost}_{k-1}(l, r, i_1 - 1, d - 1, \{+1, -1\})\]

if \(v' = \text{NULL}\) then

\[v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(\min(l, j_1 - d + 1), i_1 - 1, \{+1, -1\}, \text{left})\]

if \(v' \neq \text{NULL}\) and \(\sigma_2 \neq \sigma_1\) then

\[v' \leftarrow \text{NULL}\]

if \(v' = \text{NULL}\) then

\[v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindAtLeftmost}_{k-1}(l, r, j_1 + 1, d - 1, \{+1, -1\})\]

if \(v' = \text{NULL}\) then

\[v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(j_1 + 1, \min(i_1 + d - 1, r), \{+1, -1\}, \text{right})\]

if \(v' = \text{NULL}\) then

return \text{NULL}

else

\[v = (i_1, j_1, \sigma_1) \leftarrow \text{FindFirst}_{k-1}(t, \min(t + d - 1, r), \{+1, -1\}, \text{right})\]

if \(v = \text{NULL}\) then

return \text{NULL}

\[v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(\max(l, t - d + 1), t), \{+1, -1\}, \text{left})\]

if \(v' = \text{NULL}\) then

return \text{NULL}

if \(\sigma_1 = \sigma_2\) and \(\sigma \in s\) and \(\max(j_1, j_2) - \min(i_1, i_2) + 1 \leq d\) then

return \((\min(i_1, i_2), \max(j_1, j_2), \sigma_1)\)

else

return \text{NULL}

B. Proof of Lemma 1

The main loop of the algorithm of [9] is the following, assuming the algorithms have error at most \(1/9\):

1. for \(m = 0\) to \([\log_9 n]-1\) do:
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 \text{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
Quantum Lower and Upper Bounds for 2D-Grid and Dyck Language

\[
\sum_{n=0}^{\left\lceil \log_T n \right\rceil} 1000 \cdot O(3^n) + 1000 \cdot O(\log n) = O(3^{\log_T n}) = O(\sqrt{n/T}).
\]

C FindFirstₖ Algorithm’s Description, Complexity and Proof of Correctness

C.1 FindFixedPosₖ

Let us first describe a subroutine used by FindFirstₖ.

FindFixedPosₖ(l, r, t, s, left) searches for the leftmost substring \(x[i, j]\) such that \(\text{sign}(f(x[i, j])) \in s\) and \(|f(x[i, j])| = k\), i.e. \(i \leq t \leq j\) and there is no \(x[i', j']\) such that \(i' \leq t \leq j', i' < i\) and \(f(x[i', j']) = f(x[i, j])\).

The procedure is similar to FindAnyₖ. First, we consider a randomized algorithm that uniformly chooses \(d\) as a power of 2 that is at most \(r - l\). For this \(d\), it runs FindAtLeftmostₖ(l, r, t, d, s) algorithm and searches for a non-NULL result. The probability of getting a correct result is at least \(O(1/\log(r-l))\). Then, we apply the Amplitude amplification method and the idea from Lemma 1 that requires \(O(\sqrt{\log(r-l)})\) iterations. Similarly, we find the maximal \(d\) that finds a substring. This algorithm also performs \(O(\sqrt{\log(r-l)})\) iterations due to [13, 12]. The total complexity of the algorithm is \(O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})\) due to the complexity of FindAtLeftmostₖ.

Lemma 12. FindFixedPosₖ(l, r, t, s, left) returns the leftmost minimal substring \(x[i, j]\) such that \(\text{sign}(f(x[i, j])) \in s\) or NULL if there is no such substring. The expected running time is \(O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})\).

Proof. Let us show by induction that FindAtLeftmostₖ(l, r, t, d, s) returns the leftmost substring \(x[i, j]\) such that \(\text{sign}(f(x[i, j])) \in s\). If \(k = 2\), we check whether \(x_t = x_{t+1}\) before \(x_t = x_{t+1}\).

Assume that there is another minimal substring \(x[i', j']\) such that \(i' \leq t \leq j', f(x[i, j]) = f(x[i', j'])\) and \(i' < i\).

1. Assume that there are \(j_1\) and \(j_2\) such that \(i < j_1 < i_2 < j\), \(|f(x[i, j_1])| = |f(x[i_2, j])| = k-1\) and \(\text{sign}(f(x[i, j_1])) = \text{sign}(f(x[i_2, j])) \in s\). By induction one of the invocations of FindAtLeftmostₖ₋₁ or FindFirstₖ₋₁ finds \(x[i, j]\) and it the leftmost. Therefore, \(j' = j\). If \(i' < i\), then \(x[i', j']\) is not minimal or \(|f(x[i', j'])| > |f(x[i, j])|\), a contradiction.

2. Assume that there are \(j_1\) and \(j_2\) such that \(i < j_1 < i_2 < j\), \(|f(x[i, j_1])| = |f(x[i_2, j])| = k-1\) and \(\text{sign}(f(x[i, j_1])) = \text{sign}(f(x[i_2, j])) \in s\). By induction \(x[i_2, j]\) is the leftmost \(\pm(k-1)\) substring. Therefore, \(j' = j\). If \(i' < i\), then \(x[i', j']\) is not minimal or \(|f(x[i', j'])| > |f(x[i, j])|\), a contradiction.

If \(d > r-l\) the algorithm finds \(x[i, j]\). If \(d < r-l\), the algorithm could find the wrong substring (not the leftmost one containing \(t\)). So, we should to find the maximal \(d\) such that FindAtLeftmostₖ finds a substring. In that case, when we amplify the randomized version of the algorithm, we get the required one.

Searching by Grover’s search for the maximal \(d\) requires the same \(O(\sqrt{r-l})\) expected number of iterations due to [13, 12]. The total complexity of the algorithm is \(O(\sqrt{r-l}(\log(r-l))^{0.5(k-1)})\) due to the complexity of the FindAtLeftmostₖ procedure.
A. Ambainis et al.

FindFixedPos\(_k(l, r, t, s, \text{right})\) searches for the rightmost substring \(x[i, j]\) such that
\[
\text{sign}(f(x[i, j])) \in s \text{ and } |f(x[i, j])| = k, \text{ i.e. } i \leq t \leq j \text{ and there is no } x[i', j'] \text{ such that }
\]
\[
i' \leq t \leq j', j < j' \text{ and } f(x[i', j']) = f(x[i, j]).
\]
The algorithm is similar to FindFixedPos\(_k(l, r, t, s, \text{left})\), but uses FindAtRightmost\(_k\).

### C.2 FindFirst\(_k\) Algorithm’s Description

The FindFirst\(_k\) procedure calls FindLeftFirst\(_k\) or FindRightFirst\(_k\) depending on the direction. Since both version 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 5 FindRightFirst\(_k(l, r, s)\).** The algorithm for searching for the first ±\(k\)-substring.

\[
lBorder \leftarrow l, rBorder \leftarrow r
\]
\[
d \leftarrow 1 \quad \triangleright \text{depth of the search}
\]
\[
\text{while } lBorder + 1 < rBorder \text{ do}
\]
\[
\text{mid} \leftarrow \lfloor(lBorder + rBorder)/2\rfloor
\]
\[
v_l \leftarrow \text{FindAny}_{\pm k}(lBorder, mid, s) \quad \triangleright \text{repeat } 2d \text{ times and take the majority}
\]
\[
\text{if } v_l \neq \text{NULL then}
\]
\[
rBorder \leftarrow \text{mid}
\]
\[
\text{if } v_l = \text{NULL then}
\]
\[
v_{mid} \leftarrow \text{FindFixedPos}_{\pm k}(lBorder, rBorder, mid, s, \text{left}) \quad \triangleright \text{majority of } 2d \text{ runs}
\]
\[
\text{if } v_{mid} \neq \text{NULL then}
\]
\[
v \leftarrow v_{mid}
\]
\[
\text{Stop the loop.}
\]
\[
\text{if } v_{mid} = \text{NULL then}
\]
\[
lBorder \leftarrow \text{mid} + 1
\]
\[
d \leftarrow d + 1
\]
\[
\text{return } v
\]

### C.3 Proof of Claim on FindFirst\(_k\) Procedure from Proposition 2

Let us prove the correctness of the algorithm for \(\text{direction} = \text{right}\) 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 \(\text{mid} = \lfloor(lBorder + rBorder)/2\rfloor\). 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 \([l\text{Border}, r\text{Border}]\) segment such that \((r\text{Border} - l\text{Border})/2 \leq j - i\) 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 \([l\text{Border}, r\text{Border}]\).

The length of this segment is at most \(w \cdot 2^{-u}\) where \(w = r - l\). The complexity of \(\text{FindAny}_k(l\text{Border}, \text{mid}, s)\) is at most \(O\left(\sqrt{w \cdot 2^{-u}} \cdot \log \left(\log \left(\log \left(\log (r - l)\right)\right)^{0.5(k - 1)}\right)\right) = O\left(\sqrt{w \cdot 2^{-u}} \cdot \log \left(\log \left(\log (r - l)\right)\right)^{0.5(k - 1)}\right).

Also, \(\text{FindFixedPos}_k(l\text{Border}, r\text{Border}, \text{mid}, s, \text{left})\) has complexity \(O\left(\sqrt{w \cdot 2^{-u}} \cdot \log \left(\log \left(\log (r - l)\right)\right)^{0.5(k - 1)}\right)\).

So the total complexity of the \(u\)-th iteration is \(O\left(\sqrt{w \cdot 2^{-u}} \cdot \log \left(\log (r - l)\right)^{0.5(k - 1)}\right)\), 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:

\[
O\left(\sum_{u=1}^{\log_2 w} 2u \sqrt{w \cdot 2^{-u}} \cdot \log (r - l)^{0.5(k - 1)}\right) = O\left(\sqrt{w \cdot 2^{-u}} \cdot \log (r - l)^{0.5(k - 1)} \sum_{u=1}^{\log_2 w} u(\sqrt{2})^{-u}\right)
\]

\[
= O\left(\sqrt{w \cdot 2^{-u}} \cdot \log (r - l)^{0.5(k - 1)} \sum_{u=0}^{\infty} (u + 1)(\sqrt{2})^{-u}\right)
\]

\[
= O\left(\sqrt{w \cdot 2^{-u}} \cdot \log (r - l)^{0.5(k - 1)} \frac{\sqrt{2}}{(\sqrt{2} - 2)^2}\right)
\]

Finally, we need to analyze the success probability of the algorithm: at the \(u\)-th iteration, the algorithm will run each test \(2u\) times and each test has a constant probability of failure \(\varepsilon\). 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

\[
\left(\frac{2u}{u}\right)e^u \leq \left(\frac{2we}{u}\right)^u \leq (2ew)^u.
\]

Hence the probability that the algorithm fails is bounded by

\[
\sum_{u=1}^{\log_2 w} (2ew)^u \leq \sum_{u=1}^{\infty} (2ew)^u \leq \frac{2ew}{1 - 2ew}.
\]

By taking \(\varepsilon\) small enough (say \(2e\varepsilon < \frac{1}{4}\)), which is always possible by repeating the calls a constant number of times to boost the probability, we can ensure that the algorithm a probability of failure less than \(1/2\).

D Proof of Theorem 3

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

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

The complexity of Algorithm 3 is the same as the complexity of FINDANY$_k, n$ 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,

$$O(\sqrt{n} + 2k(\log(n + 2k))^{0.5k}) = O(\sqrt{2n(\log(2n))^{0.5k}}) = O(\sqrt{n(\log n)^{0.5k}})$$

The error probability is the same as the complexity of FINDANY$_k, n$.

## E Reduction for the proof of Theorem 4

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

$$\text{EX}_{2m}^{m+1}(x) = 0 \iff f(x) = 2$$
$$\text{EX}_{2m}^{m+1}(x) = 1 \iff f(x) = 0$$

whereas

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

If we could make sure that the minimum and maximum constraints are satisfied, DYCK$_k, n$ could be used to compute EX$_{2m}^{m+1}$. To ensure the minimum constraint, we map each block $0$ 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 EX$_{2m}^{m+1}$ where the inputs, which could now be the results of functions $\left( \text{EX}_{2m}^{m+1} \right)^{\ell-1} = x_i$, 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 $B$: $0 \leq f(x) \leq h$ and either $f(B) = 0$ or $f(B) = 2$.

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

For $\ell = 0$ (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 $\ell > 0$, let us have input bits $x = (x_1, x_2, \ldots, x_{2m})$ of EX$_{2m}^{m+1}$ satisfying the input promise. Assume that the bits (that could be equal to values of $\left( \text{EX}_{2m}^{m+1} \right)^{\ell-1}$) 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 $\text{EX}_{2m}^{m+1}(x)$ \textit{corresponds} to $B'$ because

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

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

Since for all $(w, h)$-sized blocks $B$: $\text{DYCK}_{h,w}(B) = 1 \iff f(B) = 0$ one can solve the $(\text{EX}_{2m}^{m+1})^f$ problem by running $\text{DYCK}_{h,w}$ on the \textit{corresponding} block.

See Figure 6.

**Figure 6** The reduction $\text{EX}_{2m}^{m+1} \circ \text{EX}_{2m}^{m+1} \leq \text{DYCK}_{4m+6,12m^2+2m}$. 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 Figure 1). The input word $B_1B_2 \ldots B_{2m}1^{2m}$ corresponds to the outer function $\text{EX}_{2m}^{m+1}$ with $B_i$ being a block corresponding to the output of an inner $\text{EX}_{2m}^{m+1}$. 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 $\text{EX}_{2m}^{m+1}$ to get the reduction $\text{EX}_{2m}^{m+1} \circ (\text{EX}_{2m}^{m+1})^{f-1} \leq \text{DYCK}_{O(fm),O((2m)^f)}$.

**F A quantum algorithm for Directed-2D-Connectivity$_{n,k}$**

In this section, we prove Theorem 8 by constructing a quantum algorithm for Directed-2D-Connectivity$_{n,k}$.

The main idea is to construct an AND-OR formula for Directed-2D-Connectivity$_{n,k}$ and to use the quantum algorithm for AND-OR formulae by Reichardt [15] which evaluates an AND-OR formula of size $L$ with $O(\sqrt{L})$ queries.

We first deal with the case when $n = 2^m$ for some non-negative 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 $(n/2, r)$ for some $r : 0 \leq r \leq k$. For the paths to and from $(n/2, r)$ we can apply this reasoning recursively. Let us denote by $F_{\mu,k,i,j}$ our formula for the path from vertex $(i, j)$ to $(i + 2^\mu, j + \kappa)$, and by $L_{\mu,k}$ its size (the number of variable instances it has; it does not depend on $i, j$). Thus we have the recurrent formulae

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

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

For the base case $F_{0,k,i,j}$ (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 $L_{0,k} = (\kappa + 1)^2$. 

It follows by induction on $\mu$ 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: $L_{m,k} < 2^{m+1}(e \cdot (k+m+2)/k)^k = O\left(n(e(1 + \log_2 n))^k\right)$. There exists a quantum algorithm with $O(\sqrt{L})$ queries for a formula of size $L$ [15], thus we obtain the complexity mentioned in the theorem statement.

For an arbitrary $n$ we can find the smallest $m$ for which $n \leq 2^m$ 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.