École Normale Supérieure de Lyon
Latvijas Universitāte

M1 Internship Report

Complexity of recognizing Dyck languages of bounded height with quantum query algorithms.

Key words: *Quantum Query Complexity, Dyck Words, Quantum Algorithms, Regular Languages, Star-Free Languages, Adversary Methods.*

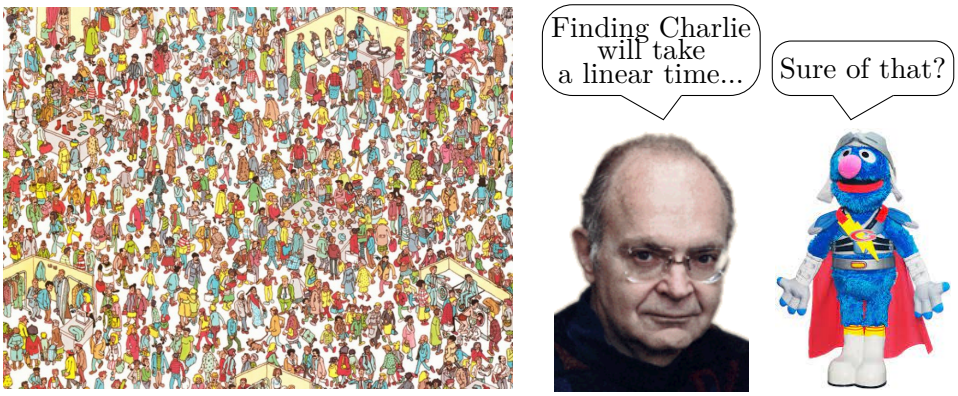

Figure 1: Historical discussion between Knuth and Grover.

Student:
Maxime Cautrès

Supervisor:
Andris Ambainis
Kamil Khadiev

May the 2nd 2022 - July the 22th 2022

Contents

1 Introduction

Context of the internship

As part of the first year of Master at the École Normale Supérieure de Lyon, I was able to do a 12 weeks research internship in a laboratory.

My research for an internship in Quantum Algorithmic had brought me to the Faculty of Computing at the University of Latvia and my supervisor Andris Ambainis. My research also brought me to discuss with Kamil Khadiev from Kazan Federal University who became my co-supervisor. We discussed by email to find an interesting subject of research on which I liked to work on. I thank them for their help, their supervision and the time they gave to me during this 12 weeks.

During the internship, I have been integrated to the life of the Center for Quantum Computing Science. I thanks members of the team for the great discussions we had after the seminar.

I also want to thank Omar Fawzi for having introduced me to quantum computing and its fascinating possibilities.

The team's research area is quantum algorithms and complexity theory. More precisely, the team works on establishing new quantum algorithms with better complexity and proving new bounds to the quantum complexity for many different types of problem belonging from graph theory to cryptography passing by language recognition theory. My work on the recognition of restricted Dyck words integrated itself great in the team work as it has already been studied by the team for few years [4] and further by Kamil Khadiev [8].

My internship, named "Complexity of recognizing Dyck language with a quantum computer", had for goal to reduce the gap between the lower and the upper bound for the quantum query complexity of recognizing Dyck words of bounded height. The best known lower and upper bounds are describe in [4] by Andris Ambainis team.

In the end of the introduction, the field of research will be presented more precisely. After that, technical preliminaries, which are useful to understand the current and the new results, will be detailed. Finally, the last two sections present my new results for the quantum query complexity of bounded height Dyck word and the different tries to improve both upper and lower bounds.

1.1 History of quantum computing

The history of quantum computing has started in 1980 when Paul Benioff, an american physicist, proposed a quantum mechanical model of Turing machines [5]. His machine uses some properties of the matter that has been discovered by quantum physicists. After that, some computer scientists suggested that the quantum model of Turing machines may be more expressive that the classical model. Few years after, the first bricks of the quantum circuit have been introduced by Richard Feynman [7]. The first quantum computers have started to arrived in the middle of 1990s. During the last 20 years, the funds given to the creation of the first quantum computer have skyrocketed, as the number of start-ups and companies dedicated to it. This emulation has made from the quantum computer field one of the most active field of research today. On the algorithmic side, the first astonishing result is the algorithm designed by Peter Shor (1994) [12]. The algorithm improves a lot the complexity of factorizing integers, enough to break our cryptographic protocols when quantum computer will be powerful enough. Since 1994, the quantum algorithm area has evolved almost independently from the quantum computers and has developed many beautiful theories and interesting results. But first, how does a quantum circuit works?

1.2 The quantum circuit, and quantum query model and complexity

In classical computer science, the piece of information is represented with 0 and 1. This two states can be easily obtained using electricity because 0 can be represented by 0V and 1 by 5V, it is easy to propagate electricity through wires and to stock its level into capacitor. Moreover a little piece of hardware, named transistor, has allowed to do some computations using logical gates which once include in a complex machine create our so-called "computers".

For quantum computers, the story isn't so different. First, the 0 and 1 are now represented using particles like electrons or photons. For example, an electron with a spin of $+\frac{1}{2}$ (noted $|1\rangle$, pronounced ket 1) represents a 1 and an other one with a spin of $-\frac{1}{2}$ (noted $|0\rangle$, pronounced ket 0) represents a 0. But the use of particles is motivated by their properties and mainly by a quantum property called superposition. A quantum state is not only $|0\rangle$ or $|1\rangle$, but can be both in the same time, i.e. $\lambda_0|0\rangle + \lambda_1|1\rangle$ for every $\lambda_{0,1} \in \mathbb{C}$ such that $|\lambda_0|^2 + |\lambda_1|^2 = 1$. A quantum bit, called qubit, correspond to a quantum state that is a superposition of two value. As before, the computations are done by gates, here quantum gates, which transform the quantum state of a qubit into another quantum state. At the end, to get the result of a computation, it is mandatory to measure the state of the quantum system, which breaks the quantum superposition. A quantum state of n qubits can be represented with a vector in a 2^n dimensional space whose norm is equal to 1, and a quantum gate by a linear unitary transformation on a 2^n dimensional space. A transformation is said unitary if it preserved the lengths.

A quantum circuit is a precise configuration of quantum gates on a finite number of qubits. The following Figure 2 represents the quantum circuit that computes a uniform randomize on $\{0,1\}^n$.

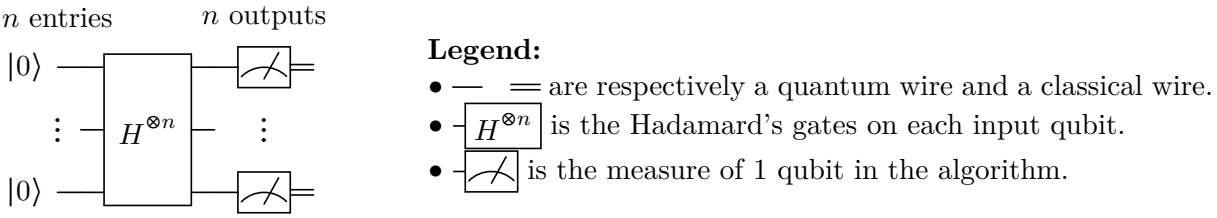

Figure 2: A quantum circuit computing the uniform random on $\{0,1\}^n$.

The quantum query circuit is a quantum circuit used to compute function on an entry $x = x_1 \dots x_N$ that belongs to an entry space. The black box model [2] of a quantum query circuit is composed of an input state $|\psi_{start}\rangle$ and a sequence $U_0, Q, U_1, \dots, Q, U_T$ of linear unitary transformations such that $|\psi_{start}\rangle$ and all U_i do not depend on entry x unlike the Q_i which depend on x. The quantum state $|\psi_{start}\rangle$ belongs to a d-dimensional space generated by $|1\rangle, |2\rangle, \dots, |d\rangle$. To define Q, the basis vectors first need to be renamed from $|1\rangle, \dots, |d\rangle$ to $|i,j\rangle$ with $i \in [\![0, N]\!]$ and $j \in [\![1, d_i]\!]$ for some d_i such that $d_1 + d_2 + \dots + d_N = d$. Next, Q is define such that

$$Q(|i,j\rangle) := \begin{cases} |0,j\rangle & \text{if } i = 0 \\ |i,j\rangle & \text{if } i > 0 \text{ and } x_i = 0 \\ -|i,j\rangle & \text{if } i > 0 \text{ and } x_i = 1. \end{cases}$$

The gates Q are doing the queries to input x by flipping some of the vectors. Finally, to get the output of the quantum query algorithm it is necessary to measure the output. A quantum query algorithm can the summarized with the following quantum circuit.

The quantum query complexity of an algorithm corresponds to the number of calls to the Q gate. Often, this number of calls is depending on the size of the entry. The quantum query

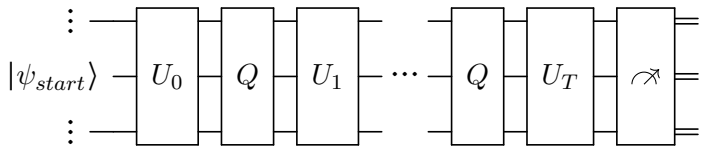

Figure 3: Structure of a quantum query algorithm.

complexity of a problem corresponds to the highest possible bound for which it is certain there is no quantum query algorithm with a lower quantum query complexity solving the problem.

1.3 Dyck Languages of height k

First, the Dyck language corresponds to the set of correct and balanced words of parenthesis. Often, computer scientist used another more graphical and convenient definition where parenthesis words are replaced with discrete paths onto a 2D space. Indeed, every path starts at coordinate $(0, 0)$ and is composed of 2 types of steps: The first one is an increasing step that is represented by adding to the end of the current path the vector $\overrightarrow{(1, 1)}$. The second one is an decreasing step, it works similarly but the add vector is $\overrightarrow{(1, -1)}$. Moreover, to be a Dyck path, the path should start with an increasing step, it should never cross the abscise axis and it should finish on it. A Dyck word of length 12 is presented in Figure 4. The Dyck language is a context free language as it can easily be recognized using a context free grammar. The work done by Andris Ambainis' team [4] focus on a restriction on Dyck language with bounded height k. More precisely, a Dyck word is of height at most k if, in every of its prefix, the difference between the number of opening and closing parenthesis does not exceeds k. In the path representation, a path is said to be of height at most k if the path never cross the line $y = k$. Figure 4 illustrates two different Dyck words with only one of height at most 3. The restricted Dyck language with bounded height k is noted DYCK_k and is interesting because it belong to the already well studied class of star free languages (Detail in subsubsection 2.1.2).

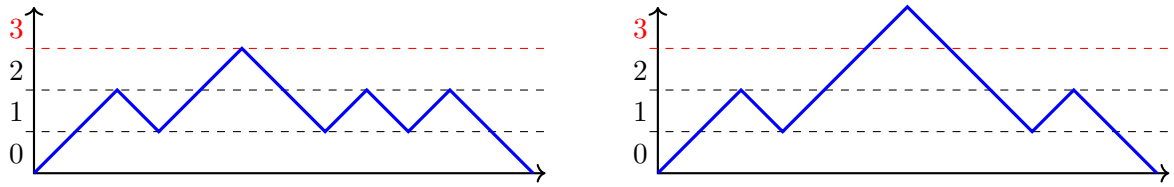

Figure 4: **On the left**, a valid Dyck word of height at most 3. **On the right**, an invalid Dyck word of height at most 3.

1.4 State of the art

This state of the art is not too precise as the understanding of the bibliography took almost the first half the internship and is more detail in section 2. First, few years ago Aaronson, Grier and Schaeffer [1, 2019] worked on quantum query complexity of recognizing regular languages as they can model a lot of tasks. They concluded that there are 3 different cases depending of the language:

- $O(1)$ if it is sufficient to read constant number of letters at the beginning and the end.
- $\tilde{\Theta}(\sqrt{n})$ if a Grover's search[1] is the best way to recognize the language. The tilde refers to the existence of a constant c_{te} such that the quantum query complexity is equal to

[1] The Grover search is a quantum query algorithm that allows to search for a marked element in a set of n elements, where p of them are marked, with a quantum query complexity of $O(\sqrt{\frac{n}{p}})$.

$\Theta\left(\sqrt{n}(\log_2(n))^{c_{te}}\right)$.

- $\Theta(n)$ if recognizing the language is the same as counting modulo some value that can be computed with $O(n)$.

Further more, it is proved that being in the second classes is equivalent to being a star-free language. Andris Ambainis' team decided to work on DYCK_k as this language is a beautiful example of star-free languages. In [4, 2020], the team focused on finding the value of c_{te}, the power of the logarithm, in order to find the exact quantum query complexity. They first proved by reduction that the quantum query complexity of DYCK_k called $Q\left(\text{DYCK}_k\right)$ is in $\Omega\left(c^k\sqrt{n}\right)$ where c is a constant greater than 1. They also gave an algorithm for DYCK_k with a quantum query complexity of $O\left(\sqrt{n}(\log(n))^{0.5k}\right)$. Since then, no better lower bound or algorithm have been found.

1.5 Goals of the internship

The problem on the quantum query complexity of DYCK_k is still open, my internship has for goal to reduce the gap between the lower bound and the best known algorithm. My researches have been organized on two main axes:

- **Increasing the lower bound.** To do this, it is necessary to understand the bibliography on the adversary method in order to try to find a new adversary with better property. An other way is to use already existing lower bounds and translates them to DYCK_k with reductions.
- **Lowering the upper bound.** It is sufficient to find new algorithms with a better quantum query complexity than the previous ones. As for lower bounds, reduction method can also provides interesting new upper bounds.

Finally, the overall goal would be to made the two bounds match in order to get the exact quantum query complexity of DYCK_k.

1.6 Results

The main results of my internship are presented in section 3. The first one is a small revision of the original quantum query algorithm that recognize DYCK_k [4] and the second one is a new quantum query algorithm for DYCK_2 and its modifications to improve more the already revised original algorithm.

2 Preliminaries

In order to understand the $\tilde{\Theta}(\sqrt{n})$ quantum query complexity, it is mandatory to learn the logic behind the trichotomy theorem [1] and its dependency on regular languages and star-free languages. After that, it will be necessary to presents the tools that computer scientists have developed in order to find new bounds on the quantum query complexity. These tools can be group into 3 main categories: reductions, algorithms, and adversary methods[13].

2.1 Quantum query for regular languages.

In the article [1], Aaronson, Grier and Schaeffer use a really interesting algebraic characterization of regular languages based on monoids and syntactic congruence. This definition was unknown to me and I spent a lot of time on understanding it correctly.

2.1.1 Regular languages

Usually, the set of regular languages $\mathcal{R}$ on the alphabet Σ is defined as the smallest fixed point of the function F that includes $\{\varnothing\} \cup \{\varepsilon\} \cup (\cup_{l \in \Sigma}\{l\})$. With F being the function that computes concatenations, unions, and Kleene stars:

$$\begin{aligned} F(X) = &\{AB,\ \forall (A,B) \in X^2\} \\ &\cup \{A \cup B,\ \forall (A,B) \in X^2\} \\ &\cup \{A \cap B,\ \forall (A,B) \in X^2\} \ \#\ \texttt{optional} \\ &\cup \{A^*,\ \forall A \in X\}. \end{aligned}$$

However in [1], the more convenient way to characterize regular languages is to use their algebraic characterization. More precisely, for every regular language L it exists a finite monoid M, a subset S of this monoid, and a monoid homomorphism δ from Σ^* to M such that the regular language is exactly the pre-image of the subset S by the monoid homomorphism δ (i.e. $\delta^{-1}(S)$). Let's take apart this characterization: A monoid is a 3-tuple of a set M, an internal associative binary operation and finally the identity element associated to the operation. A monoid homomorphism is a map from a monoid to another that preserves the operation and the identity. Now, to get every elements of the characterization, the first step is to compute the syntactic monoid. The syntactic monoid is obtained by dividing Σ^* by the following equivalence relation called syntactic congruence

$$x \sim_L y \iff \forall (u,v) \in \left(\Sigma^*\right)^2, (uxv \in L \iff uyv \in L).$$

This equivalence relation is a congruence relation as the equivalence class can be multiplied (i.e. if $x \sim_L y$ and $u \sim_L v$ then $xu \sim_L yv$). With this syntactic monoid, it is now possible to define a monoid homomorphism. For that it is sufficient to take the homomorphism that map an element to its congruence class. Moreover, using a subset of the syntactic monoid is sufficient as in a congruence class, none or every element of the class is in L. Indeed, if $x \sim_L y$ then for all (u,v) in $\left(\Sigma^*\right)^2, (uxv$ in $L \iff uyv$ in $L)$ thus x in L is equivalent to y in L. So, it exists a subset of equivalent class that represent every word of L and no word not in L. Now, the last hard things to show is that the syntactical monoid is of finite size[2]. Finally, every regular language can be recognized by finite monoid.

2.1.2 Star-free languages

The set of star-free languages is a really well studied subset of the regular languages. Its definition differs a little from regular languages' one as the Kleene star is replaced by the complement operation (noted $\overline{L}$). So star-free languages are defined as the smallest fixed point of the function F' (the function that computes concatenations, unions and complements) and such that it includes $\{\varnothing\} \cup \{\varepsilon\} \cup (\cup_{l \in \Sigma}\{l\})$. This restriction does not imply that every star-free language is finite. For example, Σ^* can be written $\overline{\varnothing}$ and the language on $\Sigma = \{1,2,3\}$ described with the regular expression $\Sigma^* 20^* 2\Sigma^*$ can be written as following in the star-free way $\overline{\varnothing 2 \overline{\varnothing} \Sigma \backslash \{0\} \overline{\varnothing}} 2 \overline{\varnothing}$.

As for regular languages, it exists an algebraic characterization for star-free languages. Let M be a monoid, M is said to be aperiodic if for every x in M it exits a positive integer n such that $x^n = x^{n+1}$. A theorem proved by Schützenberger [11] states that a language is recognized by a finite aperiodic monoid if and only if it is star-free.

[2] More precisely, the finiteness of the syntactic monoid is the main property that characterize every regular languages. A good intuition to understand is first that it is always possible to construct finite automata from a finite monoid. For the second way, it is more delicate but the work done by Brzozozowski, Szykula and Ye [6, 2018] summarized a lot of result on the influence of the size of the minimal automata size on the size of the smaller syntactic monoid.

Good examples of star-free languages are the Dyck word languages with bounded heights. First, it is easy to have a finite automaton that recognize Dyck word of height at most k by putting one state for each integer from 0 up to k. However, the belonging to the star-free regular languages is more delicate to prove. It has been done by an italian researcher in [14, 1978].

2.1.3 Trichotomy theorem

Theorem 2.1 (Aaronson, Grier and Schaeffer [1])**.** *Every regular language has a quantum query complexity* $0, \Theta(1), \tilde{\Theta}(\sqrt{n})$*, or* $\Theta(n)$ *according to the smallest class that contains the language in the following hierarchy.*

- *Degenerate: One of the four languages* $\varnothing, \varepsilon, \Sigma^*$*, or* Σ^+*.*
- *Trivial: The set of languages which have trivial* [3] *regular expressions.*
- *Star-free: The set of languages which have star-free regular expressions.*
- *Regular: The set of languages which have regular expressions.*

This theorem is really important as it gives a good idea for the quantum query complexity of many language recognitions. Moreover, the classes are now clearly defined so it is easier to know where is a problem compared to the first classification describes in the introduction. However, it does not give the exact quantum query complexity of every problem because, as said in the introduction, the result for star-free languages is given using a tilde $\sim$. The $\tilde{\Theta}(\sqrt{n})$ means that the quantum query complexity of any star-free regular language is in $\Theta(\sqrt{n}(\log_2(n))^{c_{te}})$ for some c_{te} a non negative constant. As it is known that for every k, $\textsc{Dyck}_k$ is star-free[14], it becomes an interesting problem to find the power of $\log_2(n)$ depending on the value of k.

2.2 The bounds for Dyck$_k$ problem

The trichotomy theorem state that for every k, $\textsc{Dyck}_k$ language has a quantum query complexity in $\tilde{\Theta}(\sqrt{n})$. The Θ means that the best possible algorithm is both a $\tilde{O}(\sqrt{n})$ and a $\tilde{\Omega}(\sqrt{n})$. So, a common method to find the power of $\log_2(n)$ depending on k is the squeeze theorem. More precisely, the quantum query complexity of $\textsc{Dyck}_k$ is trapped between a lower and an upper bound for the quantum query complexity. So it is possible to deduce the quantum query complexity from an increasing sequence of lower bounds and a decreasing sequence of upper bounds such that they share the same limit l with l equals to $Q(\textsc{Dyck}_k)$. How to find this sequence?

- **For the lower bounds sequence.** Some of the most important tools to compute lower bounds are called **quantum adversary methods** and have been invented by Andris Ambainis [3]. This tools can compute lower bounds more or less tight and some of this adversary methods have really useful property such as being compatible with a certain type of compositions of problems. This lead to the **reduction methods** that allow compute a lower bounds from lower bounds of different but easier problems.
- **For the upper bounds sequence.** The main method is to **find quantum query algorithms** that are more and more efficient. An other way is also **by reduction** to a more difficult problem which has a good enough upper bound.
- **For the same limit.** The idea is to do an iterative process where each step improves successively each bound until only one will continue to be improved. Finally, both bounds may end up matching. However, before my internship, both lower and upper bounds for $\textsc{Dyck}_k$ were respectively stuck to $\Omega\left(c^k\sqrt{n}\right)$ for some constant c greater than 1 and $O\left(\sqrt{n}\left(\log_2(n)\right)^{0.5k}\right)$.

[3] A language L is said to be trivial if and only if it exists 2 finite size alphabets L_1 and L_2 such that $L = L_1\Sigma^*L_2$.

2.2.1 Lower bounds on the quantum query complexity of $\mathbf{Dyck}_k$

The quantum adversary methods: This part presents some quantum adversary methods and their usages.

The adversary method. The classical adversary method is a way to prove lower bound in classical algorithmic. Indeed, the idea behind the classical adversary is to prove that for every algorithm, it exist an entry such that the algorithm cannot decide this entry in less unit of complexity than the lower bounds. In general, during the algorithm execution, the adversary modifies values of the entry that have not been already used in order to increase the execution time. This classical adversary works great for classical algorithm but is not really useable on quantum algorithms.

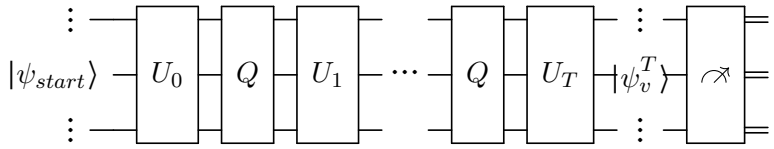

Figure 5: Recall on the structure of a quantum query algorithm.

The super basic quantum adversary method. In order to recognize a language it is mandatory to be able to distinguish between a valid word v and an invalid word w. So before the measure at the end of the quantum query algorithm, the states $|\psi_v^T\rangle$ and $|\psi_w^T\rangle$ should be distinguishable, thus $|\langle\psi_v^T|\psi_w^T\rangle| < \frac{2}{3}$. How do they will differ? The quantum query algorithm starts with $|\psi_v^0\rangle = |\psi_w^0\rangle = |\psi_{start}\rangle$. Moreover, the inner product of both states isn't affected by U_i gates because they are unitary independent of v and w. However, the Q gates affect this inner product as the behavior of each gate Q is depending on the input. Let's define the progress measure $\mathcal{P}$ such that $\mathcal{P}(t) := \langle\psi_v^t|\psi_w^t\rangle$ where $|\psi_v^t\rangle$ represents the quantum state after U_t on the entry v. Let's suppose it exists d such that for all $0 \leq t \leq T-1$, $|\mathcal{P}(t+1) - \mathcal{P}(t)| \leq d$. It implies the following inequality

$$\mathcal{P}(T) = \underbrace{\mathcal{P}(0)}_{=1} + \sum_{i=0}^{T-1} \underbrace{\mathcal{P}(i+1) - \mathcal{P}(i)}_{\geq -d} \geq 1 - T \times d. \tag{1}$$

Moreover, $\mathcal{P}(T)$ should be lower than $\frac{2}{3}$ so it implies that $1 - T \times d$ should also be lower than $\frac{2}{3}$ and finally that T should be a $O(\frac{1}{d})$. This gives a general idea about how to determined a lower bound but it isn't precise enough to get a theorem. In its courses, Ryan O'Donnell [9] explained a simple to understand adversary method called the super basic adversary method. This adversary allows to compute a lower bound by using the following method:

Theorem 2.2. *Let's define YES the set equal to $f^{-1}(accepted)$ and NO the set equal to $f^{-1}(rejected)$. If it exists two subsets $Y \subseteq YES$ and $Z \subseteq NO$ such that:*

- *For each y in Y, there are at least m strings z in Z with $dist(y,z) = 1$.*
- *For each z in Z, there are at least m' strings y in Y with $dist(y,z) = 1$.*

Then the quantum query complexity $Q(f)$ is in $\Omega(\sqrt{mm'})$.

The proof of the theorem (adapted to the definition of the quantum query circuit of the report) is detailed in Appendix A. One important result of the adversary method is the lower bound on the quantum query complexity of $Ex_{2m}^{m|m+1}$ in $\Omega(m)$. The $Ex_{2m}^{m|m+1}$ problem consists in recognizing between $|x| = 2m \ \wedge \ |x|_1 = m$, and $|x| = 2m \ \wedge \ |x|_1 = m+1$. More generally,

an adversary method define a process to find the lower value of d possible as lowering d directly increase the lower bound on the quantum query complexity (Equation 1). The super-basic adversary method as its name let suggests is simple but does not give a tight lower bounds for many problems. Two more interesting but more complex quantum adversary methods are now presented:

Let $f : D \to \{0,1\}$ be the function whose quantum query complexity is unknown.

- **The Basic adversary method by Ambainis [3].**

  Theorem 2.3. *Let's define YES the set equal to $f^{-1}(accepted)$ and NO the set equal to $f^{-1}(rejected)$. Moreover, let $Y \subseteq YES$, $Z \subseteq NO$, and let $\mathcal{R} \subseteq Y \times Z$ be a set of "hard to distinguish" pairs, such that:*

  - *For each y in Y, there are at least m strings z in Z with (y,z) in $\mathcal{R}$.*
  - *For each z in Z, there are at least m' strings y in Y with (y,z) in $\mathcal{R}$.*

  Also, let's define $\mathcal{R}_i$ as a restriction from $\mathcal{R}$ such that (y,z) is in $\mathcal{R}_i$ if and only if (y,z) is in $\mathcal{R}$ and $y_i \neq z_i$. In addition,

  - *For each y in Y and i, there are at most l strings z in Z with (y,z) in $\mathcal{R}_i$*
  - *For each z in Z and i, there are at most l' strings y in Y with (y,z) in $\mathcal{R}_i$*

  Then the quantum query complexity $Q(f)$ is in $\Omega(\sqrt{\frac{mm'}{ll'}})$.

- **The general adversary method by Reichardt [10].** A symmetric matrix Γ is an adversary matrix for f if the rows and cols of Γ can be indexed by inputs x and y in D such that $\Gamma_{x,y} = 0$ if $f(x) = f(y)$. $\Gamma^{(i)}$ is defined from Γ and is a similarly sized matrix such that $\Gamma^{(i)}_{x,y} = \begin{cases} \Gamma_{x,y} & \text{if } x_i \neq y_i \\ 0 & \text{otherwise} \end{cases}$. This two objects allow to define the following notion of adversary plus minus

$$Adv^{\pm}(f) = \max_{\substack{\Gamma \text{ - an adversary} \\ \text{matrix for } f}} \frac{\|\Gamma\|}{\max_i \|\Gamma^{(i)}\|}.$$

  Theorem 2.4. *The adversary plus minus is such that $Q(f) = \Theta(Adv^{\pm}(f))$.*

  In [10], Reichardt gave the proof that the general adversary method is not only giving a lower bound to the quantum query complexity as the method return the directly the quantum query complexity of f.

All this adversary methods are really interesting to compute lower bounds, but sometimes it is more useful to reuse already computed ones.

The reduction method: For problems, it looks obvious that some are easier than the others. Computer scientists have developed tools to handle more formally this notion of difficulty comparison. One of the main tool is named reduction. A reduction is the process to solve a first problem P_1 using an algorithm for a second one $P2$. Because the second problem is able to solve the first one, it is said to be harder, which is written $P_1 \leq P_2$.

Theorem 2.5. *Reduction and $Adv^{\pm}$*

$$P_1 \leq P_2 \implies Adv^{\pm}(P_1) \leq Adv^{\pm}(P_2)$$

Finally, problems can also be composed. Let's take $P_1 : \{0,1\}^n \to \{0,1\}$ and $P_2 : \{0,1\}^m \to \{0,1\}$. Then $P_1 \circ P_2$ is defined as

$$(P_1 \circ P_2)(x_1 \dots x_{nm}) := P_1\left(\underbrace{P_2(x_1 \dots x_m), \dots, P_2(x_{(n-1)m+1} \dots x_{nm})}_{n}\right).$$

This composition has a great behavior with the adversary plus minus.

Theorem 2.6. *Composition and $Adv^{\pm}$*

$$Adv^{\pm}(P_1 \circ P_2) \geq Adv^{\pm}(P_1) \times Adv^{\pm}(P_1)$$

Andris' team found that the $Ex_{2m}^{m|m+1}$ problem can be reduce using the OR and AND problems to $\textsc{Dyck}_k$ problem with the reduction described in [4]. They finally get a lower bound in $\Omega(c^k\sqrt{n})$ for the quantum query complexity of $\textsc{Dyck}_k$.

The team also got another result (not published), they have founded an upper bound using a reduction from $\textsc{Dyck}_k$ to a problem about the connectivity into a 2d directed grid with missing edges. This upper bound of $O(\sqrt{n}(\log_2(n))^{0.5(k-1)})$ is interesting as it is an upper bound in $\tilde{\Theta}(\sqrt{n})$.

2.2.2 Best known algorithm to recognize $\textbf{Dyck}_k$

Before checking the algorithm for $\textsc{Dyck}_k$, it is necessary to define some terms useful for later.

Preliminary definition:

- **The height function h:** This first utilitarian function allow the computation of final height of a string. It is defined as following

$$h(x_1 \dots x_n) := \sum_{i=1}^{n} (-1)^{x_i}.$$

- **$\pm k$-strings:** A string $x_1 \dots x_n$ is said to be a $+k$-string (resp. $-k$-string) if

$$\max_{1 \leq i \leq j \leq n} h(x_i \dots x_j) = k \quad \left(\text{resp. } \min_{1 \leq i \leq j \leq n} h(x_i \dots x_j) = -k\right).$$

- **minimal $\pm k$-strings:** A $\pm k$-string $x_1 ... x_n$ is said to be minimal if it doesn't exist i, j such that $1 \leq i \leq j \leq n$, $(i,j) \neq (1,n)$ with $x_i \dots x_j$ a $\pm k$-string.

$\textbf{Dyck}_k$ characterization: In order to recognize $\textsc{Dyck}_k$ language, multiple approach are possible. The method used the most by Andris Ambainis' team in [4] is to search for a substring that cannot be seen into a Dyck word of height at most k. So, a natural way to reject a word w from $\textsc{Dyck}_k$ is to search for a $\pm k+1$-string into $1^k w 0^k$. Let's detail this technic more precisely. First, a Dyck word is always above the abscise axis, so it cannot exist i such that $h(w_1 \dots w_i) = -1$, thus it cannot exist i such that $h((1^k w 0^k)_1 \dots (1^k w 0^k)_i) = -k-1$ and finally it cannot exist a $-(k+1)$-string into $1^k w 0^k$. After that, a Dyck word always end on the abscise axis, which means that it cannot exist i such that $h(w_i \dots w_n) = 1$ which implies that $1^k w 0^k$ cannot contain any $+(k+1)$-string. Moreover, for a Dyck word of height at most k, it cannot exist i such that $h(w_1 \dots w_i) = k+1$ so the bounded height constraint is already taken into account by the impossibility of having a $+(k+1)$-string into $1^k w 0^k$. Finally, the belonging of a $\pm(k+1)$-string into $1^k w 0^k$ is sufficient to reject every non Dyck word of height at most k.

$\pm(k+1)$-strings recognition: In order to recognize DYCK_k, the main point is now to find efficiently a $\pm(k+1)$-string. Let's detail a little how is it done.

- **For k=1:** To reject w, a non-DYCK_1 word, it is sufficient to find a ± 2-strings. However, there is only one minimal ± 2-strings of size 2, so every $\pm(2)$-strings can be found by searching for 11 or 00 using 2 Grover searches. This method implies a quantum query complexity of $O(\sqrt{n})$ from its two calls to Grover (Quantum query complexity of grover is in $O(\sqrt{n})$).
- **For k=2 (naive approach):** To reject, the goal is to find a $\pm(3)$-string. Unfortunately, there are an infinite number of minimal $\pm(3)$-strings as they form the language $1(10)^*11 + 0(01)^*00$. So trying every possible minimal $\pm(3)$-strings for an input string of size n require $O(n)$ calls to Grover with a resulting quantum query complexity of $O(n\sqrt{n})$. This algorithm has its complexity already above the known upper bounds of the trichotomy theorem. In order to stay into the trichotomy theorem, the algorithm should be improved.
- **For any k+1:** In order to have a faster algorithm, Ambainis' team found an inductive algorithm on the depth. Indeed, into each minimal $\pm(k+1)$-string there are two smaller minimal $\pm k$-strings as shown in Figure 6. The main ideas of the induction step are:

  1. Choose an upper bound d in $\{2, 4, 8, \ldots, 2^{\lfloor \log_2(n) \rfloor}\}$ for the length of the $\pm(k+1)$-string.
  2. Choose an indices t in $\{1, 2, 3, \ldots, n\}$ that has to be in the possible $\pm(k+1)$-string.
  3. Try to find two $\pm(k)$-strings in an interval of length at most d that include t
     (a) Try to find a $\pm(k)$-string that include t of length at most d-1.
     (b) If it exists, find an other $\pm(k)$-string on the left or on the right, if it fail return `NULL`.
     (c) If it does not exist, try to find $\pm(k)$-string on the left, and another $\pm(k)$-string on the right, if it fail return `NULL`.
     (d) Test if both $\pm(k)$-strings are of the same sign and if the string that include both if of length lower than d.
     (e) If the test is good , return the $\pm(k+1)$-string, otherwise return `NULL`.

  Let's explain quantum query complexity and the ideas behind each step. First, it has been shown before that searching for every minimal $\pm(k+1)$-string isn't a solution as it is too slow. So, the idea is to bound the size of the minimal $\pm(k+1)$-string the function is currently searching for. This is interesting as it allows to use a function describes in step 3 named $\text{FINDATLEFTMOST}_{k+1}$, whose main parameters are d and t, and which is able to found and return a minimal $\pm(k+1)$-string if and only if the returned substring includes the index t and has its size between $\frac{d}{2}$ and d. These constraints imply two things:

  – First, the parameter t implies a call to Grover (as it may be possible to have a $\pm(k+1)$-string not including t), but there are $O(d)$ values of t such that $\text{FINDATLEFTMOST}_{k+1}$ return a minimal $\pm(k+1)$-string so it is possible to cut early the Grover search to get its quantum query complexity in $O\left(\sqrt{\frac{n}{d}}\right)$. This Grover search corresponds to item 2 named $\text{FINDFIXEDLENGTH}_{k+1}$.
  – After that for d, in order not to miss any minimal $\pm(k+1)$-string, it is necessary to call $\text{FINDFIXEDLENGTH}_{k+1}$ for every d in $\{2, 4, 8, \ldots, 2^{\lfloor \log_2(n) \rfloor}\}$ (item 1) which implies a call to Grover in $O\left(\sqrt{\log(n)}\right)$. This is done in item 1 by the function named FINDANY_{k+1}.

  Finally, the quantum query complexity of $\text{FINDATLEFTMOST}_{k+1}$ is $O\left(\sqrt{d}(\log_2(d))^{0.5(k-1)}\right)$, it comes from complex calls to many subroutines. In steps 3a and 3b, at most three calls to

FindAtLeftMost$_k$ are done with a quantum query complexity of $O\left(\sqrt{d}(\log_2(d))^{0.5(k-2)}\right)$. In step 3b and 3c, at most 4 calls to FindFirst$_k$ are done. The subroutine FindFirst$_k$ find the closer $\pm k$-string from the end r or the beginning l of a specified interval by doing a binary search using calls to FindAny$_k$ and FindFixedPos$_k$ in a quantum query complexity of $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$. This last subroutine FindFixedPos$_k$ searches only for $\pm k$-strings which include the index t, it has a quantum query complexity of $O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$.

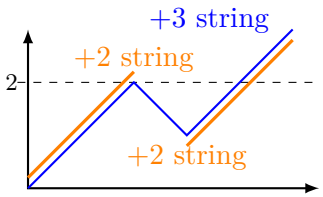

Figure 6: Decomposition of a +3-string into two $\pm(k+1)$-strings.

Finally, this complex algorithm has a quantum query complexity of $O\left(\sqrt{n}(\log(n))^{0.5k}\right)$ which is a $\tilde{\Theta}\left(\sqrt{n}\right)$. This description of the existing algorithm stay at the surface in order to stay simple, every subroutines' pseudo code can be found in Appendix B. No proof of their quantum query complexities are provided as a proof for Theorem 3.2, almost identical, is already provided in Appendix C. However, this algorithm can be improved, indeed the upper bound by reduction to 2d directed grid ($O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$) is still better than the quantum query complexity of the algorithm ($O\left(\sqrt{n}(\log_2(n))^{0.5k}\right)$).

3 A better algorithm for Dyck$_k$

3.1 A better complexity analysis of the original algorithm

In the article [4], Andris Ambainis's team gave a quantum query algorithm to recognize Dyck$_k$ that uses $O(\sqrt{n}(\log_2(n))^{0.5k})$ quantum queries. But the quantum query complexity for $k=1$ ($O\left(\sqrt{n\log_2(n)}\right)$) is not as good as a Grover's search $O(\sqrt{n})$ which is sufficient (seen in subsubsection 2.2.2). More precisely, the logarithmic search done by FindAny$_{k+1}$ for $k=1$ is useless as there is exactly one minimal ± 2 -string. So, it is sufficient to add a new initial case to FindAny$_{k+1}$ for $k=1$ in order to use a Grover search for 00 and 11 in $1w0$ instead of FindFixedLength$_2$. This small revision has lowered the quantum query complexity for $k=1$ of the function to $O(\sqrt{n})$ instead of $O(\sqrt{n\log_2(n)})$. This gives the following algorithm for FindAny$_k$

Algorithm 1 FindAny$_k$(l,r,s)

Require: $0 \leq l < r$ and $s \subseteq \{1,-1\}$
  if $k > 2$ **then**
    Find d in $\{2^{\lceil\log_2(k)\rceil}, 2^{\lceil\log_2(k)+1\rceil}, \ldots, 2^{\lceil\log_2(r-l)\rceil}\}$ such that
      $v_d \leftarrow$ FindFixedLength$_k(l,r,d,s)$ is **not** Null
    return v_d or Null if none
  else
    Find t in $\{l, l+1, \ldots, r\}$ such that
      $v_t \leftarrow$ FindAtLeftMost$_2(l,r,t,2,s)$ is **not** Null
    return v_t of Null if none.

The same improvements can be done on FindFixedPos$_k$ because if $k=2$ the logarithmic

search is useless. So FINDFIXEDPOS$_k$ can be redefined as in ALGORITHM2. For $k = 2$, the complexity is lowered from $O(\sqrt{\log_2(l-r)})$ to $O(1)$.

Algorithm 2 FINDFIXEDPOS$_k(l, r, t, s)$

Require: $0 \leq l < r$, $l \leq t \leq r$ and $s \subseteq \{1, -1\}$
  if $k > 2$ **then**
    Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
      $v_d \leftarrow$ FINDATLEFTMOST$_k(l, r, t, d, s)$ is **not** NULL
    return v_d or NULL if none
  else $v \leftarrow$ FINDATLEFTMOST$_k(l, r, t, 2, s)$ is **not** NULL
    return v_d or NULL if none

This small improvements on two initial cases improve the global quantum query complexity of each subroutine and finally the quantum query complexity for DYCK$_k$.

Theorem 3.1. Dyck$_k$'s algorithm correctness *The new definitions of* FINDANY *and* FINDFIXEDPOS *do not change the behavior the original algorithm as other subroutines (Appendix B) stay unchanged.*

Proof Theorem 3.1. The behavior of the DYCK$_k$ algorithm with the new subroutines hasn't changed as FINDANY (resp. FINDFIRST) has the same sub-behavior on every entry than its former definition.

□

Theorem 3.2. Dyck$_k$'s subroutines complexity *The quantum query complexity of* DYCK$_k$ *algorithm's subroutines are the following.*

1. $Q($DYCK$_k) = O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$ *for* $k \geq 1$
2. $Q($FINDANY$_{k+1}(l, r, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ *for* $k \geq 1$
3. $Q($FINDFIXEDLENGTH$_{k+1}(l, r, d, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
4. $Q($FINDATLEFTMOST$_{k+1}(l, r, t, d, s)) = \begin{cases} O\left(\sqrt{d}(\log_2(d))^{0.5(k-2)}\right) & \text{for } k \geq 2 \\ O(1) & \text{for } k = 1 \end{cases}$
5. $Q($FINDFIRST$_k(l, r, s, left)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \geq 2$
6. $Q($FINDFIXEDPOS$_k(l, r, t, s)) = \begin{cases} O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) & \text{for } k \geq 3 \\ O(1) & \text{for } k = 2 \end{cases}$

Proof Theorem 3.2. The idea is that only the $O(\sqrt{n})$ comes from the initial case for $k = 1$ and that each of the $k-1$ level of the recursion increases the final quantum query complexity with a $O\left(\sqrt{\log_2(n)}\right)$ factor. The $O\left(\sqrt{\log_2(n)}\right)$ factor is proven by Andris Ambainis' team in [4] while the $O(\sqrt{n})$ for $k = 1$ comes from the new version of FINDANY$_k$ (ALGORITHM 2). The complete proof for the theorem is given in Appendix C.

□

Unfortunately, the improvements done on the initial cases of some of the subroutines are not sufficient to get a significant improvement for the quantum query complexity of DYCK$_k$ algorithm. However, they are sufficient to match the theoretical upper bound gave by the reduction.

3.2 A new algorithm for Dyck_2

First, we would like to find an algorithm with a quantum query complexity near to match the lower bound describes by Andris Ambainis' team in [4], $\exists c > 1$ such that $Q(\text{DYCK}_k) = \Omega\left(\sqrt{n}c^k\right)$. So the searched algorithm may have a quantum query complexity of $O(\sqrt{n})$.

For $k = 1$, the query complexity comes only from a call to Grover's search. For $k = 2$ it no more possible to search for every minimal ± 3-strings without a linear number of call to Grover's search. So in order to keep the quantum query complexity in $O(\sqrt{n})$, the algorithm should do a constant number of calls to Grover's search.

For that, we define a new alphabet that can express every even length binary string and that has convenient properties for a Grover's search. Let $\mathcal{A} = \{a, b, c, d\}$ be the alphabet where a corresponds to 00, b to 11, c to 01, and d to 10. Every string of size 2 has its associated letter in $\mathcal{A}$ thus every even length bit string is expressed in $\mathcal{A}^*$. This alphabet allows to prove more easily the following theorem.

Theorem 3.3. ***Substrings rejection for Dyck word of height at most 2.*** *A word on the alphabet $\mathcal{A}$ embodies a Dyck word of height at most 2 if and only if it does not contain $aa, ac, bb, bd, cb, cd, da,$ ordc as a substring.*

Proof Theorem 3.3. First, this alphabet $\mathcal{A}$ is important because each letter has a height variation in $\{-2, 0, 2\}$. Indeed, a has a height variation of 2, b of -2, c of 0, and d of 0. So, after each letter in a word, the current height will be even. Moreover, for a valid Dyck word of height at most 2, only 0 and 2 are even and accessible heights. So after every letter in a word, the height will be 0 or 2 which are respectively the lower and the upper bound for the height.

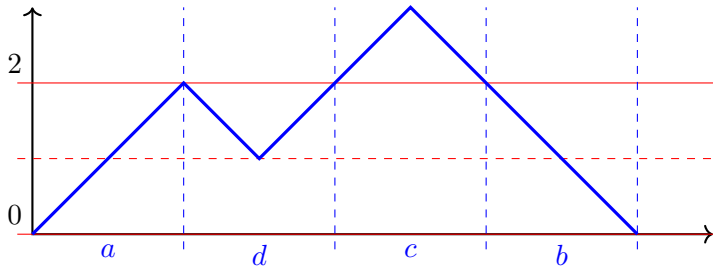

Figure 7: Illustration of the letters of $\mathcal{A}$ using Dyck's representation.

This property is important as it implies that every substrings that cross a bounds uses at least two letters. So it may be possible that the belonging of some pairs of letters to a word implies its non belonging to DYCK_2. It founds out that $\mathcal{A}^2$ can be split into two sets described in Table 1.

Table 1: Partition of $\mathcal{A}$ into $\mathcal{X}, \mathcal{V}$.

$\mathcal{X}$	*aa ac bb bd cb cd da dc*
$\mathcal{V}$	*ab ad ba bc ca cc db dd*

- The set $\mathcal{X}$. First, every couple of letters which contains a ± 3-string is in $\mathcal{X}$. This first condition explains the belonging of $aa, ac, dc, da, cb, bb, bd,$ and cd. Next, cd and dc belong to $\mathcal{X}$. Indeed, for any valid Dyck word of height at most 2, the current height is bounded between 0 and 2. After each letter the current height is even so both couple cd and dc start and finish on the same bound. However, cd and dc are going above and below the height at which they start so both are going outside off the bounds. Thus, a word which contains cd or dc can not be a Dyck Word of height a most 2. The Figure 8 shows each couple of $\mathcal{X}$.

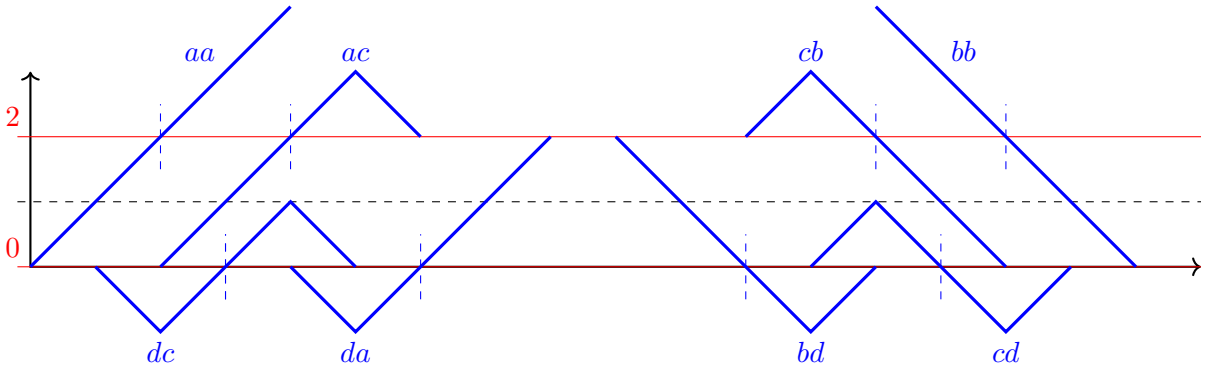

Figure 8: Every 2 letters configuration that implies that a word, whom a configuration is a substring, is not a Dyck word of height at most 2.

- The set $\mathcal{V}$. Every couple of $\mathcal{A}$ does not imply that the word is not a Dyck word of height at most 2 because some couple of $\mathcal{A}$ can fit inside the height bounds. The Figure 9 shows that every couple not in $\mathcal{X}$ (ie. $ab, ad, ba, bc, ca, cc, db$, and dd) can fit between heights 0 and 2.

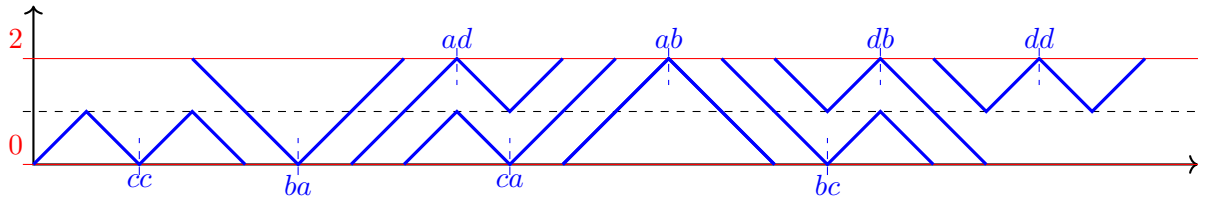

Figure 9: Every 2 letters configuration that can be found in a valid Dyck word of height at most 2.

So, a word in $\mathcal{A}^*$ that has a substring in $\mathcal{X}$ cannot be a Dyck word of height two. But does every non Dyck word of height at most 2 has a substring in $\mathcal{X}$?

A word is not a Dyck word of height at most 2 if it include a ± 3-string. But how are represented ± 3-strings using the letters? There are 8 different cases which are 2 by 2 symmetrical so Figure 10 and Figure 11 show only the cases for $+3$-strings. In Figure 10, every $+3$-string of size 3 is include in aa or ac so it is sufficient to search for this two couples. In Figure 11 every $+3$-string of length greater than 3 is composed of 2 minimal $+2$-strings. It implies that one must be a a while the other must be da of dc. Because da or dc are rejecting substrings, it is sufficient to search for them.

□

Finally, a word on the alphabet $\mathcal{A}$ embodies a Dyck word of height at most 2 if and only if it does not contain $aa, ac, bb, bd, cb, cd, da, dc$. The following Algorithm 3 for Dyck_2 comes from the direct application of the theorem.

Theorem 3.4. ***The quantum query complexity of*** $\textbf{DyckFast}_{2,n}$. *The* $\text{DyckFast}_{2,n}$ *algorithm has a quantum query complexity of* $O\left(\sqrt{n}\right)$.

Proof Theorem 3.4. The algorithm is doing at most 8 Grover's searches on the modified input string $11x00$. So the total quantum query complexity is

$$Q(\text{DyckFast}_{2,n}) = 8 \times O\left(\sqrt{n+4}\right) = O\left(\sqrt{n}\right).$$

□

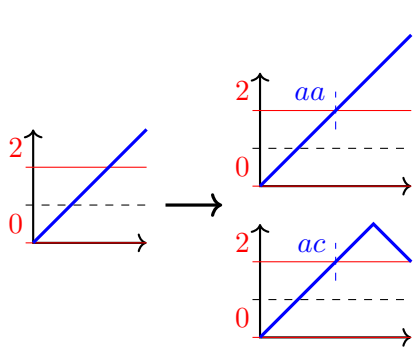

Figure 10: Configuration for a +3-strings of size 3.

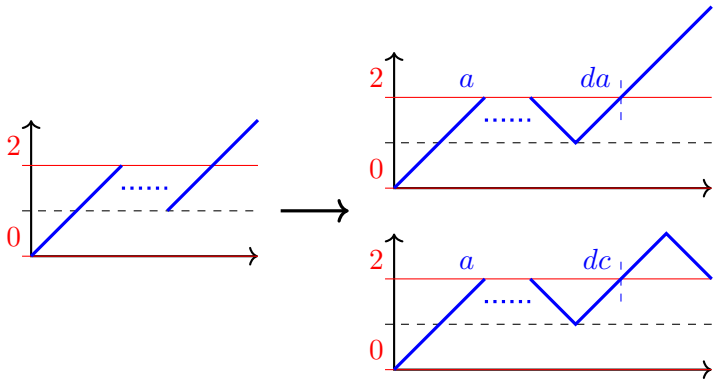

Figure 11: Configurations for a +3-string of size greater than 3.

Algorithm 3 DyckFast$_{2,n}$

Require: $n \geq 0$, x such that $|x| = 2n$
  $x \leftarrow 11x00$
  `t` $\leftarrow$ Null
  for `reject_symbol` $\in \{aa, ac, bb, bd, cb, cd, da, dc\}$ **do**
    if t == Null **then**
      Find `t` in $[\![0, n]\!]$ such that
      $x[2\texttt{t} + 1, \ldots, 2\texttt{t} + 4] =$ `reject_symbol`
  return `t` == Null

3.3 A simplification for Dyck$_2$ algorithm

The algorithm describes in subsection 3.2 is complex to explain but it turns out that a simpler version of it exists and is really easier to understand that the first one.

Algorithm 4 DyckFastEasy$_{2,n}$

Require: $n \geq 0$, x such that $|x| = 2n$
  $x \leftarrow 11x00$
  Find `t` in $[\![1, n + 1]\!]$ such that
    $x[2\texttt{t}, 2\texttt{t} + 1] \in \{00, 11\}$
  return `t` == Null

Lets explain how it works. A Dyck word of height at most 2 is recognized by the following grammar on the alphabet $\{0, 1\}$:

$$S \mapsto 0I1$$
$$I \mapsto 01I|10I|\varepsilon$$

Indeed, a Dyck word of height at most 2 $w_1 \ldots w_n$ can only start with a zero else it gets a negative height. After, if it goes up, it reach the height 2 so it can only go down. If it goes down instead of going up, it reaches height 0 so it can only go up. This implies that after every letter of odd indices, the height is equal to one, and that the next two letters can only be in $\{01, 10\}$. So, finding a 00, or 11 starting on an even index is equivalent to not being a dyck word of height at most two. Finally, it becomes sufficient to use two Grover searches for 00 and 11 in order to recognize Dyck$_{2,n}$.

3.4 A final improvement to Dyck$_k$ algorithm

The last algorithm describes is useful to recognize DYCK$_2$. However, it cannot be plug into the huge one because it does not return the ± 3-string that allows it to reject. Indeed, the algorithm only finds the second ± 2-string of the ± 3-string. So,in order to reduce the complexity of the main algorithm, it is required to find the first ± 2-string. To do that, it is sufficient to make a call to the already existing subroutines FINDFIRST$_2$ described in 9 by asking for the closest ± 2-string on the left of the already known $\pm 2-$string. This is done with a quantum query complexity of $O(\sqrt{n})$ (Computations are similar to the ones done in the proof in Appendix C). Thus, it is easy to modify a little the algorithm DYCKFASTEASY2,n to return the ± 3-string when it rejects. Finally, it can be plug into the main algorithm, the computation of the quantum query complexity are almost identical to the one done in Appendix C but initial case is for $k = 2$ instead of $k = 1$. The consequence is the removing of a recursion step which implies the loss of a $\sqrt{\log_2(n)}$ factor which bring the quantum query complexity of the algorithm that recognizing DYCK$_k$ to $O(\sqrt{n}(\log_2(n))^{0.5(k-2)})$.

4 Multiple attempts to improve the quantum query complexity upper and lower bounds

4.1 An attempt to expand the new algorithm's first version to every k

The second version of the fast algorithm is a lot simpler than the previous one, but it doesn't make the first version of the algorithm obsolete. Indeed, the first algorithm was thought with the idea of being easily modified in order to work with higher value of k. Unfortunately, the key property that make the algorithm works (i.e. after each letter with an even index, the height is equal to 0 or 2, both being the bounds for the height) no more holds. So I didn't succeed to generalized to greater value of k as the behaviors of intermediate height become too complex.

4.2 An attempt for a new algorithm for any k

To reduce the quantum query complexity of DYCK$_k$, one idea was to reduce the number of recursive calls during the execution. For that, two different possibilities seemed possible, both using the same main principle: The recursive calls could be associated to nested loops, whose variables would correspond to the choice of d for each recursion level. Indeed, in every recursion level, FINDANY$_k$ is called by FINDANY$_{k+1}$ with l and r such that $r - l \leq d_{k+1}$ with d_{k+1} the value give to d in the call from FINDANY$_{k+1}$. It is in reality more complicated but the goal is just to check if, first it is possible to compute the quantum query complexity of this model, and after to translate the method to get a better algorithm to recognize DYCK$_k$.

A natural graph structure on the (k-2)-tuples. The initial cases of the recursive algorithm are for k equal to 2, 3, so there are only $k-2$ nested loops, with one d_i for each recursive level. This $k-2$ variables can be consider together as $k-2$-tuples. In every tuple, the inequality $d_{k+1} \geq d_k \geq \ldots \geq d_4$ is verified and each tuple is a vertex in a graph (V, E). The edges E are defined as following.

$$(t_u, t_v) \in E \Leftrightarrow \exists! i \in \{1, k-2\}, ((t_u)_i \neq (t_v)_i \wedge (t_v)_i = 2(t_u)_i)$$

A vertex of the graph is said to me marked if and only if

> it exists a $\pm(k+1)$-strings of length at most $(t_u)_1$ composed of 2 $\pm k$-strings of size at most $(t_u)_2$ composed them selves of 2 $\pm(k-1)$-strings of size at most $(t_u)_3$, ... and composed themselves of 2 ± 3-strings of size at most $(t_u)_k - 2$.

This graph has an interesting property $\mathcal{P}$ for every vertex t_u. If t_u is marked then every descendent t_v of t_u in the graph is also marked. Without using this property, the use of this $k-2$ nested loops is equivalent to a DFS that search for a marked vertex. Then the quantum query complexity of this algorithm would be the one of the quantum DFS multiplied by the one to check a marked vertex. It gives us a quantum query complexity in $O\left(\sqrt{\log_2(n)^{k-2}} \times Q(\text{mark checking})\right)$. The quantum query complexity of checking if a vertex is marked depends a lot on what is allowed to do and the final quantum query complexity required. For the nested loops, the goal is to have an algorithm that improves the quantum query complexity of $O\left(\sqrt{n}\log_2(n)^{0.5(k-2)}\right)$. Thus, the quantum query complexity of checking a vertex should be at most $O(\sqrt{n})$. But what does it mean to check a vertex? This is difficult to define as the current model of nested loop is quite far from the original algorithm and not really finish to define. But, one can do some hypothesis and state that as for the original algorithm, it is only possible to check easily ($O(\sqrt{n})$) a second type of marks (which is not compatible with the property $\mathcal{P}$) define as

> it exists a $\pm(k+1)$-strings of length between $(t_u)_1/2$ and $(t_u)_1$ composed of 2 $\pm k$-strings of size between $(t_u)_2/2$ and $(t_u)_2$ composed them selves of 2 $\pm(k-1)$-strings of size between $(t_u)_3/2$ and $(t_u)_3$, ... and composed themselves of 2 ± 3-strings of size between $(t_u)_{k-2}$ and $(t_u)_{k-2}$.

With this statement, it is not possible to use an algorithm whose probability to check a node can be zero otherwise, as for the algorithm by Andris Ambainis's team, it is not possible to be sure enough of the answer. The only solutions are graph traversal that have a continuous border between explored and non explored vertices (DFS or BFS for example) and that becomes faster using Grover. So, in order to go around this constraint, it may be possible to use a slower algorithm to check the first type of mark if it implies a non negligible improvement on the quantum query complexity of the traversal as it is not really clear what would be the quantum query complexity of checking the first type of marks (not well defined).

In order to present the main way to reduce the cost of the traversal, it is useful to order the vertex of the graph onto a discrete finite space of dimension $k-2$ were every axis has a logarithmic scale (base 2) starting from 1 to $2^{\lfloor \log_2(n) \rfloor}$.

In this space, a vertex t_u has its coordinate on axis i equal to $(t_u)_i$. The goal here is to do a binary search on each dimension of the $k-2$-tuple in order to find a marked vertex. In fact, the property $\mathcal{P}$ implies that if a vertex t_u is marked then every vertex t_v is marked if t_u and t_v differ only on the j-th dimension and $(t_u)_j \leq (t_v)_j$. So on every of the $k-2$ dimensions, it is possible to do a binary search because it exists a value, depending on the values of all the first dimensions such that no vertex is marked before and all vertices are marked after. Thus, the quantum query complexity of this multidimensional binary search would be

$$O\left(\sqrt{\log_2(\log_2(n))}^{k-2} \times Q(\text{marking checking})\right).$$

The improvement is quite spectacular if it is possible to find an algorithm that check the mark of a vertex with a quantum query complexity that does not counter balance the benefit. With more time, it would have been possible to define more precisely what it means to check a mark and may be an algorithm could have been found. It may also be possible to prove that such an algorithm cannot exits.

4.3 A attempt to find a new adversary plus minus for Dyck_k

The idea was first to search for a basic adversary (Theorem 2.3) for small value of k. The efforts were focus on $k = 2$. The goal was to find two sets of word from $\{0,1\}^n$ that would have satisfied the requirements of the basic adversary. Moreover, this two sets should give a lower bound of

the form $O(\sqrt{n}f(n))$ (for some function $f(n)$) in order to increase the current lower bound of $O(\sqrt{n}c^2)$ (for some constant c). But, this basic adversary returns lower bounds of the form $\sqrt{\frac{mm'}{ll'}}$ where m, m', l, and l' (depending on n) specify properties on both sets. I do not have found such sets, more precisely, it is already difficult to think about a way to define two sets such that their sizes increase nicely according to a function in n. So, having to deal in the same time with their associated values m, m', l, and l' make the task really difficult.

4.4 An attempt to find a new reduction from easier problems

In order to find a new lower bound, one idea was to try new reductions. For $k = 2$, some languages have been tested but an optimal algorithm has been found before it worked. For $k = 3$, one of the most interesting language is defined by $c^*(ac^*bc^*)^*$. In order to reject a word w, it is sufficient to search if a substring v from bwa can be part of the language $ac^*a + bc^*b$. So, this language is also a star-free language (the same arguments than the one for DYCK_k would work here). It means that its quantum query complexity is in $\tilde{\Theta}(\sqrt{n})$. Moreover, it is possible to reduce this problem to DYCK_3 simply by doing the rewriting operations $a \mapsto 00, b \mapsto 11$ and $c \mapsto 01$. So, finding the quantum query complexity of this problem can be useful in order to get a new lower bound on the quantum query complexity of DYCK_3.

5 Conclusion

The work I led, during the internship, in order to get the exact quantum query complexity of DYCK_k, didn't allow me to merge both upper and lower bounds. However, the multiple attempts used to improve both bounds have been half successful. Indeed, every attempt to use new reductions or to compute new adversary lower bounds didn't succeed. Nonetheless, the upper bound have been improved two times, from $O(\sqrt{n}(\log_2(n))^{0.5k})$ to $O(\sqrt{n}(\log_2(n))^{0.5(k-1)})$ and finally to $O(\sqrt{n}(\log_2(n))^{0.5(k-2)})$, thanks to a small revision of the original algorithm and to a faster algorithm for DYCK_2.

For the future, there is still a lot of things to try. First, it may be possible to prove that the approach (detailed in subsection 4.2) to simplified the quantum query algorithm cannot work. Next, it could be possible that the next breakthrough for the quantum query complexity of DYCK_k comes from a totally different quantum query algorithm. Finally, Andris team's members have found a potential candidate solution for a general adversary of DYCK_k, unfortunately they didn't succeed to compute its adversary bound.

This internship has allowed me to experience the life in an other country and in a new laboratory. The discussions with different colleagues about their research and mine were really interesting and have helped me to understand more precisely the scope and the main problems of quantum computing.

References

[1] Scott Aaronson, Daniel Grier, and Luke Schaeffer. A quantum query complexity trichotomy for regular languages, 2018.

[2] Andris Ambainis. Understanding quantum algorithms via query complexity.

[3] Andris Ambainis. Quantum lower bounds by quantum arguments, 2000.

[4] Andris Ambainis, Kaspars Balodis, Jānis Iraids, Kamil Khadiev, Vladislavs Kļevickis, Krišjānis Prūsis, Yixin Shen, Juris Smotrovs, and Jevgēnijs Vihrovs. Quantum lower and upper bounds for 2d-grid and dyck language. *Leibniz International Proceedings in Informatics*, 170, 2020.

[5] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, (22(5):653-591), 1980.

[6] Janusz A. Brzozowski, Marek Szykula, and Yuli Ye. Syntactic complexity of regular ideals. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. Theory of Computing Systems, 2018.

[7] Richard Feymann. Quantum mechanical computers. *Foundamentale Physics*, (16, 507-531), 1986.

[8] Kamil Khadiev and Dmitry Kravchenko. Quantum algorithm for dyck language with multiple types of brackets. In Irina Kostitsyna and Pekka Orponen, editors, *Unconventional Computation and Natural Computation - 19th International Conference, UCNC 2021, Espoo, Finland, October 18-22, 2021, Proceedings*, volume 12984 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2021.

[9] Ryan O'Donnell. The adversary method: Lecture 20 of quantum computation at cmu. 2018.

[10] Ben W. Reichardt. Span programs and quantum query complexity: The general adversary bound is nearly tight for every boolean function. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, oct 2009.

[11] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Inf. Control.*, 8:190–194, 1965.

[12] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[13] Robert Spalek and Mario Szegedy. All quantum adversary methods are equivalent. 2004.

[14] Crespi-Reghizzi Stefano, Guida Giovanni, and Mandrioli Dino. Noncounting context-free languages. 1978.

6 Appendix

List of Figures

List of Algorithms

The frame of the internship

My internship took place into the quantum computing team of the latvian university and was under the supervision of Andris Ambainis, the team leader. The team was composed of at most 10 active members and was organized around one team seminar every wednesday. Because of the covid restriction that ended one week before the beginning of my internship, a lot of the researched have continued to work at home during my internship. So, I never really discuss with researcher until I was admitted to weekly seminar after 6 weeks. Other researchers were really welcoming and they gave their time to talk with me about the subject. I have never see any of the phd students of the team.

About Andris Ambainis, he is an incredible researcher and is remarks during the seminar were always incredible. His supervision was quite strange. He never speaks first or revives a discussion, I have taken every initiative myself. It has direct consequence on the supervision. I think I have really experiment the work of a researcher, where my supervisor was just a distant colleague if I was not insisting a lot to get small helps. I think it comes from the fact that there is no tradition of internship there so he has done what he does with PHD students. However, if a student want to do an internship with Andris, I would just warm him or her of this because if he or she is really independent at work and curious about life in Latvia, he or she will spend a really interesting and good internship with Andris in Riga.

Finally, Latvia is a small, poor and beautiful country with incredibly open minded people, spending time there have made me change my mind on many subject of every day life.

A Proof of the super-basic adversary method

As a little reminder here is the theorem.

Theorem A.1. *Let define YES the set equal to $f^{-1}(accepted)$ and NO the set equal to $f^{-1}(rejected)$. If it exist two subset $Y \subseteq YES$ and $Z \subseteq NO$ such that:*

- *For each y in Y, there are at least m strings z in Z with $dist(y,z) = 1$.*
- *For each z in Z, there are at least m' strings y in Y with $dist(y,z) = 1$.*

Then the quantum query complexity $Q(f)$ is in $\Omega(\sqrt{m \times m'})$

Understanding this proof is important as it gives the intuitions to understand the general adversary.

Proof Theorem A.1. First, let's define $\mathcal{R} := \{(y,z)$ such that $y \in Y, z \in Z$, and $\text{dist}(y,z) = 1\}$. Now, it is necessary to redefine the progress notion such that $\mathcal{P}(t) := \sum_{(y,z)\in\mathcal{R}} |\langle\psi_y^t|\psi_z^t\rangle|$ where $|\psi_x^t\rangle$ is the state of the algorithm after t query to the entry x. So, $\mathcal{P}(0) = |\mathcal{R}|$ and $\mathcal{P}(T) = |\frac{2}{3}\mathcal{R}|$. In order to prove the theorem, lets try to prove that for all t,

$$\mathcal{P}(t) - \mathcal{P}(t+1) \leq \frac{2}{\sqrt{mm'}}|\mathcal{R}|$$

as it is a correct value of d such that the lower bound is $O(\sqrt{mm'})$. Moreover, there are relations between sizes of the sets:

$$|\mathcal{R}| \geq m|Y|, |\mathcal{R}| \geq m'|Z| \implies |2\mathcal{R}| \geq m|Y| + m'|Z|.$$

Thus, instead of proving $\mathcal{P}(t) - \mathcal{P}(t+1) \leq \frac{2}{\sqrt{mm'}}|\mathcal{R}|$, lets try to prove

$$\mathcal{P}(t) - \mathcal{P}(t+1) \leq \frac{1}{\sqrt{mm'}}(m|Y| + m'|Z|) = \sqrt{\frac{m}{m'}}|Y| + \sqrt{\frac{m'}{m}}|Z|.$$

Now, lets take any a couple (y,z) in $\mathcal{R}$, it exists a unique i^* such that $y_{i^*} \neq z_{i^*}$. This small variation between y and z can be well translated to the progress after quantum query gate. Lets write $|\psi_y^t\rangle$ and $|\psi_z^t\rangle$:

$$|\psi_y^t\rangle = \sum_{\substack{i\in[\![0,N]\!] \\ j\in[\![1,d_i]\!]}} \alpha_{i,j}|i,j\rangle \qquad |\psi_z^t\rangle = \sum_{\substack{i\in[\![0,N]\!] \\ j\in[\![1,d_i]\!]}} \beta_{i,j}|i,j\rangle.$$

Now by applying Q_t, the states become

$$|\psi_y^t\rangle = \sum_{\substack{i\in[\![0,N]\!] \\ j\in[\![1,d_i]\!]}} (-1)^{(0y)_{i+1}}\alpha_{i,j}|i,j\rangle \qquad |\psi_z^t\rangle = \sum_{\substack{i\in[\![0,N]\!] \\ j\in[\![1,d_i]\!]}} (-1)^{(0z)_{i+1}}\beta_{i,j}|i,j\rangle.$$

However, y and z only differ on the index i^*, so the restricted progress measure to y and z at t and $t+1$ are equal to

$$\mathcal{P}_{y,z}(t) = \sum_{\substack{i\in[\![0,N]\!]\\ j\in[\![1,d_i]\!]}} \overline{\alpha_{i,j}}\beta_{i,j} \qquad \mathcal{P}_{y,z}(t+1) = \sum_{\substack{i\in[\![0,N]\!]\\ j\in[\![1,d_i]\!]}} (-1)^{(0y)_{i+1}+(0z)_{i+1}}\overline{\alpha_{i,j}}\beta_{i,j}$$
$$= \sum_{\substack{i\in[\![0,N]\!]\\ j\in[\![1,d_i]\!]}} \overline{\alpha_{i,j}}\beta_{i,j} - 2\sum_{j\in[\![1,d_{i^*}]\!]} \overline{\alpha_{i^*,j}}\beta_{i^*,j}$$
$$= \mathcal{P}_{y,z}(t) - 2\sum_{j\in[\![1,d_{i^*}]\!]} \overline{\alpha_{i^*,j}}\beta_{i^*,j}.$$

So, for restricted progress measures there is the inequality

$$|\mathcal{P}_{y,z}(t)| - |\mathcal{P}_{y,z}(t+1)| \le |\mathcal{P}_{y,z}(t) - \mathcal{P}_{y,z}(t+1)|$$
$$\le |2\sum_{j\in[\![1,d_{i^*}]\!]} \overline{\alpha_{i^*,j}}\beta_{i^*,j}|$$
$$\le 2\sum_{j\in[\![1,d_{i^*}]\!]} |\alpha_{i^*,j}||\beta_{i^*,j}|.$$

Moreover, the math trick for all p, q real numbers and h non negative real number $2pq \le hp^2 + \frac{1}{h}q^2$ allow to rewrite the previous inequality to

$$|\mathcal{P}_{y,z}(t)| - |\mathcal{P}_{y,z}(t+1)| \le 2\sum_{j\in[\![1,d_{i^*}]\!]} |\alpha_{i^*,j}||\beta_{i^*,j}|$$
$$\le \sum_{j\in[\![1,d_{i^*}]\!]} h|\alpha_{i^*,j}|^2 + \frac{1}{h}|\beta_{i^*,j}|^2.$$

By chosing h equal to $\sqrt{\frac{m}{m'}}$, it becomes

$$|\mathcal{P}_{y,z}(t)| - |\mathcal{P}_{y,z}(t+1)| \le \sum_{j\in[\![1,d_{i^*}]\!]} \sqrt{\frac{m}{m'}}|\alpha_{i^*,j}|^2 + \sqrt{\frac{m'}{m}}|\beta_{i^*,j}|^2.$$

Now, it is time to sum on every couple (y,z) of $\mathcal{R}$. But i^* is depending on y, z as much as $\alpha_{i,j}$s and $\beta_{i,j}$s so lets give them y and z as argument

$$\mathcal{P}(t) - \mathcal{P}(t+1) = \sum_{(y,z)\in\mathcal{R}} \left|\mathcal{P}_{y,z}(t)\right| - \sum_{(y,z)\in\mathcal{R}} \left|\mathcal{P}_{y,z}(t+1)\right|$$
$$\le \sum_{\substack{(y,z)\in\mathcal{R}\\ j\in[\![1,d_{i^*_{y,z}}]\!]}} \sqrt{\frac{m}{m'}}|\alpha^y_{i^*_{y,z},j}|^2 + \sqrt{\frac{m'}{m}}|\beta^z_{i^*_{y,z},j}|^2.$$

Now, it is useful to do the first summation in two part in order to find a upped bound to the

sum

$$\mathcal{P}(t) - \mathcal{P}(t+1) \leq \sqrt{\frac{m}{m'}} \sum_{\substack{y \ st \ \exists z \\ (y,z)\in\mathcal{R}}} \sum_{\substack{z \\ (y,z)\in\mathcal{R}}} \sum_{\substack{\text{width} \\ j\in[\![1,d_{i^*_{y,z}}]\!]}} |\alpha^y_{i^*_{y,z},j}|^2 + \sqrt{\frac{m'}{m}} \sum_{\substack{z \ st \ \exists y \\ (y,z)\in\mathcal{R}}} \sum_{\substack{y \\ (y,z)\in\mathcal{R}}} \sum_{\substack{\text{width} \\ j\in[\![1,d_{i^*_{y,z}}]\!]}} |\beta^z_{i^*_{y,z},j}|^2.$$

Now, at y fixed, $z \mapsto i^*_{y,z}$ is an injective function from $\{z, (y,z) \in \mathcal{R}\}$ to $[\![1,N]\!]$ as y and z are at distance 1. Thus $(z,j) \mapsto (i^*_{y,z}, j)$ is also an injective function from $\{(z,j), (y,z) \in \mathcal{R}$ and $j \in [\![1, d_{i^*_{y,z}}]\!]\}$ to $[\![1,N]\!] \times [\![1, d_{i^*_{y,z}}]\!]$. So, it implies that

$$\bigsqcup_{\substack{z \ \text{width} \\ (y,z)\in\mathcal{R}}} \{i^*_{y,z}\} \times [\![1, d_{i^*_{y,z}}]\!] \subseteq \bigsqcup_{n\in[\![0,N]\!]} \{n\} \times [\![1, d_n]\!]$$

and finally that

$$\sum_{\substack{z \ \text{width} \\ (y,z)\in\mathcal{R}}} \sum_{j\in[\![1,d_{i^*_{y,z}}]\!]} |\alpha^y_{i^*_{y,z},j}|^2 \leq \sum_{i\in[\![0,N]\!]} \sum_{j\in[\![1,d_i]\!]} |\alpha^y_{i,j}|^2 \leq \left\langle \psi^t_y | \psi^t_y \right\rangle \leq 1.$$

By symmetry it also works for the second sum, so the difference of progress is now the following

$$\begin{aligned}
\mathcal{P}(t) - \mathcal{P}(t+1) \leq & \sqrt{\frac{m}{m'}} \sum_{\substack{y \ st \ \exists z \\ (y,z)\in\mathcal{R}}} \sum_{\substack{z \\ (y,z)\in\mathcal{R}}} \sum_{\substack{\text{width} \\ j\in[\![1,d_{i^*_{y,z}}]\!]}} |\alpha^y_{i^*_{y,z},j}|^2 \\
& + \sqrt{\frac{m'}{m}} \sum_{\substack{z \ st \ \exists y \\ (y,z)\in\mathcal{R}}} \sum_{\substack{y \\ (y,z)\in\mathcal{R}}} \sum_{\substack{\text{width} \\ j\in[\![1,d_{i^*_{y,z}}]\!]}} |\beta^z_{i^*_{y,z},j}|^2 \\
\leq & \sqrt{\frac{m}{m'}} \sum_{\substack{y \ st \ \exists z \\ (y,z)\in\mathcal{R}}} 1 + \sqrt{\frac{m'}{m}} \sum_{\substack{z \ st \ \exists y \\ (y,z)\in\mathcal{R}}} 1 \\
\leq & \sqrt{\frac{m}{m'}} |Y| + \sqrt{\frac{m'}{m}} |Z| \\
\leq & \frac{2}{\sqrt{mm'}} |\mathcal{R}|.
\end{aligned}$$

To conclude, it gives a lower bound $\Omega(\sqrt{mm'})$ for the quantum query complexity of f.

□

B The algorithm for Dyck$_{k,n}$

All the subroutines' pseudo codes can be found from Algorithm 5 to Algorithm 10.

Algorithm 5 Dyck$_{k,n}$

Require: $n \geq 0$ and $k \geq 1$
Ensure: $|x| = n$
  $x \leftarrow 1^k x 0^k$
  $v \leftarrow \text{FindAny}_{k+1}(0, n + 2 * k - 1, \{1, -1\})$
  return v = Null

Algorithm 6 FindAny$_k(l, r, s)$

Require: $0 \leq l < r$ and $s \subseteq \{1, -1\}$
  Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
    $v_d \leftarrow \text{FindFixedLength}_k(l, r, d, s)$ is **not** Null
  return v_d or Null if none

Algorithm 7 FindFixedLength$_k(l, r, d, s)$

Require: $0 \leq l < r$, $1 \leq d \leq r - l$ and $s \subseteq \{1, -1\}$
  Find t in $\{l, l + 1, \ldots, r\}$ such that
    $v_t \leftarrow \text{FindAtLeftMost}_k(l, r, t, d, s)$ is **not** Null
  return v_t of Nullif none

Algorithm 8 FindAtLeftMost$_k(l, r, d, t, s)$

Require: $0 \leq l < r$, $l \leq r \leq r$, $1 \leq d \leq r - l$ and $s \subseteq \{1, -1\}$
  $v = (i_1, j_1, \sigma_1) \leftarrow \text{FindAtLeftMost}_{k-1}(l, r, t, d - 1, \{1, -1\})$
  if $v \neq$ Null **then**
    $v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindAtRightMost}_{k-1}(l, r, i_1 - 1, d - 1, \{1, -1\})$
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(\max(l, j_1 - d + 1), i_1 - 1, \{1, -1\}, left)$
    if $v' \neq$ Null and $\sigma_2 \neq \sigma_1$ **then** $v' \leftarrow$ Null
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindAtLeftMost}_{k-1}(l, r, j_1 + 1, d - 1, \{1, -1\})$
    if $v' =$ Null **then**
      $v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(j_1 + 1, \max(r, i_1 + d - 1), \{1, -1\}, right)$
    if $v' =$ Null **then return** Null
  else
    $v = (i_1, j_1, \sigma_1) \leftarrow \text{FindFirst}_{k-1}(t, min(t + d - 1, r), \{1, -1\}, right)$
    if v = Null **then return** Null
    $v' = (i_2, j_2, \sigma_2) \leftarrow \text{FindFirst}_{k-1}(max(t - d + 1, l), t, \{1, -1\}, left)$
    if $v' =$ Null **then return** Null
  if $\sigma_1 = \sigma_2$ and $\sigma_1 \in s$ 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 Null

Algorithm 9 $\textsc{FindFirst}_k(l, r, s, left)$

Require: $0 \le l < r$ and $s \subseteq \{1, -1\}$
  $lBorder \leftarrow l, rBorder \leftarrow r, d \leftarrow 1$
  while $lBorder + 1 < rBorder$ **do**
    $mid \leftarrow \lfloor (lBorder + rBorder)/2 \rfloor$
    $v_l \leftarrow \textsc{FindAny}_k(lBorder, mid, s)$
    if $v_l \neq \textsc{Null}$ **then** $rBorder \leftarrow mid$
    else
      $v_{mid} \leftarrow \textsc{FindFixedPos}_k(lBorder, rBorder, mid, s, left)$
      if $v_{mid} \neq \textsc{Null}$ **then return** v_{mid}
      else $lBorder \leftarrow mid + 1$
    $d \leftarrow d + 1$
  return Null

Algorithm 10 $\textsc{FindFixedPos}_k(l, r, t, s)$

Require: $0 \le l < r$, $l \le t \le r$ and $s \subseteq \{1, -1\}$
  Find d in $\{2^{\lceil \log_2(k) \rceil}, 2^{\lceil \log_2(k)+1 \rceil}, \ldots, 2^{\lceil \log_2(r-l) \rceil}\}$ such that
    $v_d \leftarrow \textsc{FindAtLeftMost}_k(l, r, t, d, s)$ is **not** Null
  return v_d or Null if none

C The proof of the quantum query complexity for $\textsc{Dyck}_{k,n}$ algorithm's subroutines

Theorem C.1. $\textsc{Dyck}_{k,n}$'s *Subroutines complexity* *The subroutines' quantum query complexity for k are the following.*

1. $Q(\textsc{Dyck}_{k,n}) = O\left(\sqrt{n}(\log_2(n))^{0.5(k-1)}\right)$ *for* $k \ge 1$
2. $Q(\textsc{FindAny}_{k+1}(l, r, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ *for* $k \ge 1$
3. $Q(\textsc{FindFixedLength}_{k+1}(l, r, d, s)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \ge 2$
4. $Q(\textsc{FindAtLeftMost}_{k+1}(l, r, t, d, s)) = \begin{cases} O\left(\sqrt{d}(\log_2(d))^{0.5(k-2)}\right) & \text{for } k \ge 2 \\ O(1) & \text{for } k = 1 \end{cases}$
5. $Q(\textsc{FindFirst}_k(l, r, s, left)) = O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ *for* $k \ge 2$
6. $Q(\textsc{FindFixedPos}_k(l, r, t, s)) = \begin{cases} O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) & \text{for } k \ge 3 \\ O(1) & \text{for } k = 2 \end{cases}$

Proof Theorem C.1. The proof is done by induction on the height k of the Dyck word.

Initialization: For $k = 1$ and $k = 2$ we have the following initialization.

- For $k = 1$, only $\textsc{FindAtLeftMost}_2$, $\textsc{FindAny}_2$,and $\textsc{Dyck}_{1,n}$ are defined. The $O(1)$ quantum query complexity of $\textsc{FindAtLeftMost}_2$ comes directly from the definition of its initial case, as the $O(\sqrt{r-l})$ quantum query complexity of $\textsc{FindAny}_2$. Then the $O(\sqrt{n})$ quantum query complexity of $\textsc{Dyck}_{1,n}$ comes from the call to $\textsc{FindAny}_2$.

- For $k = 2$, the inductive part of the algorithm starts and every subroutines is defined. The $O(1)$ quantum query complexity of $\textsc{FindFixedPos}_2$ comes from the call to $\textsc{FindAtLeftMost}_2$. The $O\left(\sqrt{r-l}\right)$ quantum query complexity of $\textsc{FindFirst}_2$ comes from the dichotomize search using $\textsc{FindAny}_2$ and $\textsc{FindFixedPos}_2$ because $\sum_{u=1}^{log_2(r-l)} 2u\left(O\left(\sqrt{\frac{r-l}{2^{u-1}}}\right)+O(1)\right) = O(\sqrt{r-l})$ (Detailed in the induction). The $O(\sqrt{d})$ quantum query complexity of $\textsc{FindAtLeftMost}_3$ comes from the constant amount of calls to $\textsc{FindFirst}_2$ and $\textsc{FindAtLeftMost}_2$ with entry of size d. The $O(\sqrt{r-l})$ quantum query complexity of $\textsc{FindFixedLength}_3$ comes from the $O\left(\sqrt{\frac{r-l}{d}}\right)$ calls to $\textsc{FindAtLeftMost}_3$. The $O\left(\sqrt{(r-l)\log_2(r-l)}\right)$ quantum query complexity of $\textsc{FindAny}_3$ comes from the $O\left(\sqrt{\log_2(r-l)}\right)$ calls to $\textsc{FindFixedLength}_3$. Finally, the $O\left(\sqrt{(r-l)\log_2(r-l)}\right)$ quantum query complexity of $\textsc{Dyck}_2$ comes from the call to $\textsc{FindAny}_3$.

Induction: Lets suppose it exists k such that Theorem 3.2 is true for k. Lets prove that it is true for $k+1$.

First, the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ quantum query complexity of $\textsc{FindFixedPos}_{k+1}$ comes from the $O\left(\sqrt{\log(r-l)}\right)$ calls to $\textsc{FindAtLeftMost}_{k+1}$.

$$\begin{aligned}
Q(\textsc{FindFixedPos}_{k+1}(l,r,t,s)) &= O(\sqrt{\log(r-l)}) \times O\left(Q(\textsc{FindAtLeftMost}_{k+1}(l,r,t,d,s))\right) \\
&\overset{IH}{=} O\left(\sqrt{\log(r-l)} \times \sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right) \\
&= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)
\end{aligned}$$

Thus the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-2)}\right)$ quantum query complexity of $\textsc{FindFirst}_{k+1}$ comes from the dichotomize search using calls to $\textsc{FindAny}_{k+1}$ and $\textsc{FindFirst}_{k+1}$.

$$\begin{aligned}
Q(\textsc{FindFirst}_{k+1}(l,r,t,d,s)) = &\sum_{u=1}^{\log_2(r-l)} 2u \times O\left(Q(\textsc{FindAny}_{k+1}(0,\tfrac{r-l}{2^{u-1}},s))\right) \\
&+ \sum_{u=1}^{\log_2(r-l)} 2u \times O\left(Q(\textsc{FindFixedPos}_{k+1}(0,\tfrac{r-l}{2^{u-1}},_,s,left))\right) \\
\overset{IH}{=}\ & O\left(\sum_{u=1}^{\log_2(r-l)} 2u \times \sqrt{\frac{r-l}{2^{u-1}}}(\log_2(\frac{r-l}{2^{u-1}}))^{0.5(k-1)}\right) \\
=\ & O\left(\sum_{u=1}^{\log_2(r-l)} 2u \times \sqrt{\frac{r-l}{2^{u-1}}}(\log_2(r-l))^{0.5(k-1)}\right) \\
=\ & O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)} \sum_{u=1}^{\log_2(r-l)} u \times (\frac{1}{\sqrt{2}})^{u-1}\right) \\
=^{a}\ & O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)} \frac{\sqrt{2}^2}{(\sqrt{2}-1)^2}\right) \\
=\ & O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)
\end{aligned}$$

[a] $\sum_{u=1}^{+\infty}\left(\frac{\mathrm{d}}{\mathrm{d}x}(x^u)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{\mathrm{d}}{\mathrm{d}x}\left(\sum_{u=1}^{+\infty} x^u\right)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{\sqrt{2}}\right) \leq \left(\frac{1}{(1-x)^2}\right)\left(\frac{1}{\sqrt{2}}\right) \leq \frac{1}{(1-\frac{1}{\sqrt{2}})^2} \leq \frac{\sqrt{2}^2}{(\sqrt{2}-1)^2}$

Next, the $O\left(\sqrt{d}(\log_2(d))^{0.5(k-1)}\right)$ quantum query complexity comes of $\textsc{FindAtLeftMost}_{k+2}$from the constant amount of calls to $\textsc{FindAtLeftMost}_{k+1}$, $\textsc{FindAtRightMost}_{k+1}$,and $\textsc{FindFirst}_{k+1}$.

$$\begin{aligned} Q(\textsc{FindAtLeftMost}_{k+2}(l,r,t,d,s)) = \; & 3 \times O\left(Q(\textsc{FindAtLeftMost}_{k+1}(l,r,t,d,\{1,-1\}))\right) \\ & +4 \times O\left(Q(\textsc{FindFirst}_{k+1}(l,r,\{1,-1\},left))\right) \\ \overset{IH}{=} \; & O\left(\sqrt{d}(\log_2(d))^{0.5(k-1)}\right) \end{aligned}$$

After that, the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right)$ quantum query complexity of $\textsc{FindFixedLength}_{k+2}$ comes from the $O\left(\sqrt{\frac{r-l}{d}}\right)$ calls to $\textsc{FindAtLeftMost}_{k+2}$.

$$\begin{aligned} Q(\textsc{FindFixedLength}_{k+2}(l,r,d,s)) &= O\left(\sqrt{\frac{r-l}{d}}\right) \times O\left(Q(\textsc{FindAtLeftMost}_{k+2}(l,r,t,d,s))\right) \\ &= O\left(\sqrt{\frac{r-l}{d}} \times \sqrt{d}(\log_2(d))^{0.5(k-1)}\right) \\ &= O\left(\sqrt{r-l}(\log_2(d))^{0.5(k-1)}\right) \\ &= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right) \end{aligned}$$

Hence, the $O\left(\sqrt{r-l}(\log_2(r-l))^{0.5k}\right)$ quantum query complexity of $\textsc{FindAny}_{k+2}$ comes from the $O\left(\sqrt{\log_2(r-l)}\right)$ calls to $\textsc{FindFixedLength}_{k+2}$.

$$\begin{aligned} Q(\textsc{FindAny}_{k+2}(l,r,s)) &= O\left(\sqrt{\log(r-l)}\right) \times O\left(Q(\textsc{FindFixedLength}_{k+2}(l,r,d,s))\right) \\ &= O\left(\sqrt{\log(r-l)} \times \sqrt{r-l}(\log_2(r-l))^{0.5(k-1)}\right) \\ &= O\left(\sqrt{r-l}(\log_2(r-l))^{0.5k}\right) \end{aligned}$$

Finally, the $O\left(\sqrt{n}(\log_2(n))^{0.5k}\right)$ quantum query complexity of $\textsc{Dyck}_{k+1,n}$ comes from the call to $\textsc{FindAny}_{k+2}$.

$$\begin{aligned} Q(\textsc{Dyck}_{k+1,n}) &= O\left(Q(\textsc{FindAny}_{k+2}(0,n+2k+1,s))\right) \\ &= O\left(Q(\textsc{FindAny}_{k+2}(0,n,s))\right) \\ &= O\left(\sqrt{n}(\log_2(n))^{0.5k}\right) \end{aligned}$$

Conclusion: By the induction principle, the Theorem 3.2 is true for $k \in \mathbb{N}^*$

□