Extended Learning Graphs for Triangle Finding*

Titouan Carette[1], Mathieu Laurière[2], and Frédéric Magniez[3]

1 **Ecole Normale Supérieure de Lyon, Lyon, France**
titouan.carette@ens-lyon.fr
2 **NYU-ECNU Institute of Mathematical Sciences at NYU Shanghai, Shanghai, China**
mathieu.lauriere@nyu.edu
3 **CNRS, IRIF, Univ Paris Diderot, Paris, France**
frederic.magniez@cnrs.fr

Abstract

We present new quantum algorithms for Triangle Finding improving its best previously known quantum query complexities for both dense and sparse instances. For dense graphs on n vertices, we get a query complexity of $O(n^{5/4})$ without any of the extra logarithmic factors present in the previous algorithm of Le Gall [FOCS'14]. For sparse graphs with $m \geq n^{5/4}$ edges, we get a query complexity of $O(n^{11/12}m^{1/6}\sqrt{\log n})$, which is better than the one obtained by Le Gall and Nakajima [ISAAC'15] when $m \geq n^{3/2}$. We also obtain an algorithm with query complexity $O(n^{5/6}(m \log n)^{1/6} + d_2\sqrt{n})$ where d_2 is the variance of the degree distribution.

Our algorithms are designed and analyzed in a new model of learning graphs that we call extended learning graphs. In addition, we present a framework in order to easily combine and analyze them. As a consequence we get much simpler algorithms and analyses than previous algorithms of Le Gall *et al* based on the MNRS quantum walk framework [SICOMP'11].

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, F.1.1 Models of Computation

Keywords and phrases Quantum query complexity, learning graphs, triangle finding

Digital Object Identifier 10.4230/LIPIcs.STACS.2017.20

1 Introduction

Decision trees form a simple model for computing Boolean functions by successively reading the input bits until the value of the function can be determined. In this model, the *query complexity* is the number of input bits queried. This allows us to study the complexity of a function in terms of its structural properties. For instance, sorting an array of size n can be done using $O(n \log n)$ comparisons, and this is optimal for comparison-only algorithms.

In an extension of the deterministic model, one can also allow randomized and even quantum computations. Then the speed-up can be exponential for partial functions (*i.e.* problems with promise) when we compare deterministic with randomized computation, and randomized with quantum computation. The case of total functions is rather fascinating. For them, the best possible gap can only be polynomial between each models [20, 4], which is still useful in practice for many problems. But surprisingly, the best possible gap is still an open question, even if it was improved for both models very recently [3, 1]. In the context of quantum computing, query complexity captures the great algorithmic successes of quantum

* This work has been partially supported by the European Commission project Quantum Algorithms (QALGO) and the French ANR Blanc project RDAM.

computing like the search algorithm of Grover [13] and the period finding subroutine of Shor's factoring algorithm [22], while at the same time it is simple enough that one can often show tight lower bounds.

Reichardt [21] showed that the general adversary bound, formerly just a lower bound technique for quantum query complexity [14], is also an upper bound. This characterization has opened an avenue for designing quantum query algorithms. However, even for simple functions it is challenging to find an optimal bound. Historically, studying the query complexity of specific functions led to amazing progresses in our understanding of quantum computation, by providing new algorithmic concepts and tools for analyzing them. Some of the most famous problems in that quest are Element Distinctness and Triangle Finding [10]. Element Distinctness consists in deciding if a function takes twice the same value on a domain of size n, whereas Triangle Finding consists in determining if an n-vertex graph has a triangle. Quantum walks were used to design algorithms with optimal query complexity for Element Distinctness. Later on, a general framework for designing quantum walk based algorithms was developed with various applications [18], including for Triangle Finding [19].

For seven years, no progress on Triangle Finding was done until Belovs developed his beautiful model of *learning graphs* [5], which can be viewed as the minimization form of the general adversary bound with an additional structure imposed on the form of the solution. This structure makes learning graphs easier to reason about without any background on quantum computing. On the other hand, they may not provide always optimal algorithms. Learning graphs have an intuitive interpretation in terms of electrical networks [7]. Their complexity is related to the total conductance of the underlying network and its effective resistance. Moreover this characterization leads to a generic quantum implementation based on a quantum version of random walks which is time efficient and preserves query complexity.

Among other applications, learning graphs have been used to design an algorithm for Triangle Finding with query complexity $O(n^{35/27})$ [6], improving on the previously known bound $\tilde{O}(n^{1.3})$ obtained by a quantum walk based algorithm [19]. Then the former was improved by another learning graph using $O(n^{9/7})$ queries [16]. This learning graph has been proven optimal for the original class of learning graphs [9], known as *non-adaptive learning graphs*, for which the conductance of each edge is constant. Then, Le Gall showed that quantum walk based algorithms are indeed stronger than non-adaptive learning graphs for Triangle Finding by constructing a new quantum algorithm with query complexity $\tilde{O}(n^{5/4})$ [11]. His algorithm combines in a novel way combinatorial arguments on graphs with quantum walks. One of the key ingredient is the use of an algorithm due to Ambainis for implementing Grover Search in a model whose queries may have variable complexities [2]. Le Gall used this algorithm to average the complexity of different branches of its quantum walk in a quite involved way. In the specific case of sparse graphs, those ideas have also demonstrated their advantage for Triangle Finding on previously known algorithms [12].

The starting point of the present work is to investigate a deeper understanding of learning graphs and their extensions. Indeed, various variants have been considered without any unified and intuitive framework. For instance, the best known quantum algorithm for k-Element Distinctness (a variant of Element Distinctness where we are now checking if the function takes k times the same value) has been designed by several clever relaxations of the model of learning graphs [5]. Those relaxations led to algorithms more powerful than non-adaptive learning graphs, but at the price of a more complex and less intuitive analysis. In **Section 3**, we extract several of those concepts that we formalize in our new model of *extended learning graphs* (**Definition 3.1**). We prove that their complexity (**Definition 3.2**) is always an upper bound on the query complexity of the best quantum algorithm solving

the same problem (**Theorem 3.3**). We also introduce the useful notion of *super edge* (**Definition 3.4**) for compressing some given portion of a learning graph. We use them to encode efficient learning graphs querying a part of the input on some given index set (**Lemmas 3.7 and 3.8**). In some sense, we transpose to the learning graph setting the strategy of finding all 1-bits of some given sparse input using Grover Search.

In **Section 4**, we provide several tools for composing our learning graphs. We should first remind the reader that, since extended learning graphs cover a restricted class of quantum algorithms, it is not possible to translate all quantum algorithms in that model. Nonetheless we succeed for two important algorithmic techniques: Grover Search with variable query complexities [2] (**Lemma 4.1**), and Johnson Walk based quantum algorithms [19, 18] (**Theorem 4.2**). In the last case, we show how to incorporate the use of super edges for querying sparse inputs.

We validate the power and the ease of use of our framework on Triangle Finding in **Section 5**. First, denoting the number of vertices by n, we provide a simple adaptive learning graph with query complexity $O(n^{5/4})$, whose analysis is arguably much simpler than the algorithm of Le Gall, and whose complexity is cleared of logarithmic factors (**Theorem 5.1**). This also provides a natural separation between non-adaptive and adaptive learning graphs. Then, we focus on sparse input graphs and develop extended learning graphs. All algorithms of [9] could be rephrased in our model. But more importantly, we show that one can design more efficient ones. For sparse graphs with $m \geq n^{5/4}$ edges, we get a learning graph with query complexity $O(n^{11/12}m^{1/6}\sqrt{\log n})$, which improves the results of [12] when $m \geq n^{3/2}$ (**Theorem 5.2**). We also construct another learning graph with query complexity $O(n^{5/6}(m \log n)^{1/6} + d_2\sqrt{n})$, where d_2 is the variance of the degree distribution (**Theorem 5.3**). To the best of our knowledge, this is the first quantum algorithm for Triangle Finding whose complexity depends on this parameter d_2.

2 Preliminaries

We will deal with Boolean functions of the form $f : Z \to \{0,1\}$, where $Z \subseteq \{0,1\}^N$. In the query model, given a function $f : Z \to \{0,1\}$, the goal is to evaluate $f(z)$ by making as few queries to the z as possible. A query is a question of the form 'What is the value of z in position $i \in [N]$?', to which is returned $z_i \in \{0,1\}$.

In this paper we will discuss functions whose inputs are graphs viewed through their adjacency matrice. Then z will encode an undirected graph G on vertex set $[n]$, that is $N = \binom{n}{2}$ in order to encode the possible edges of G. Then $z_{ij} = 1$ iff ij is an edge of G.

In the quantum query model, these queries can be asked in superposition. We refer the reader to the survey [15] for precise definitions and background on the quantum query model. We denote by $Q(f)$ the number of queries needed by a quantum algorithm to evaluate f with error at most $1/3$. Surprisingly, the general adversary bound, that we define below, is a tight characterization of $Q(f)$.

▶ **Definition 2.1.** Let $f : Z \to \{0,1\}$ be a function, with $Z \subseteq \{0,1\}^N$. The *general adversary bound* $\mathrm{Adv}^{\pm}(f)$ is defined as the optimal value of the following optimization problem:

$$\text{minimize: } \max_{z \in Z} \sum_{j \in [N]} X_j[z,z] \quad \text{subject to: } \sum_{j \in [N] \,:\, x_j \neq y_j} X_j[x,y] = 1, \text{ when } f(x) \neq f(y),$$
$$X_j \succeq 0, \ \forall j \in [N],$$

where the optimization is over positive semi-definite matrices X_j with rows and columns labeled by the elements of Z, and $X_j[x,y]$ is used to denote the (x,y)-entry of X_j.

▶ **Theorem 2.2** ([14, 17, 21])**.** $Q(f) = \Theta(\mathrm{Adv}^{\pm}(f))$.

3 Extended learning graphs

Consider a Boolean function $f : Z \to \{0,1\}$, where $Z \subseteq \{0,1\}^N$. The set of positive inputs (or instances) will be usually denoted by $Y = f^{-1}(1)$. A 1-*certificate* for f on $y \in Y$ is a subset $I \subseteq [N]$ of indices such that $f(z) = 1$ for every $z \in Z$ with $z_I = y_I$, where $z_I = (z_i)_{i \in I}$.

3.1 Model and complexity

Intuitively, learning graphs are simply electric networks of a special type. The network is embedded in a rooted directed acyclic graph, which has few similarities with decision trees. Vertices are labelled by subsets $S \subseteq [n]$ of indices. Edges are basically from any vertex labelled by, say, S to any other one labelled $S \cup \{j\}$, for some $j \notin S$. Such an edge can be interpreted as querying the input bit x_j, while x_S has been previously learnt. The weight on the edge is its conductance: the larger it is, the more flow will go through it. Sinks of the graph are labelled by potential 1-certificates of the function we wish to compute. Thus a random walk on that network starting from the root (labelled by $\emptyset$), with probability transitions proportional to conductances, will hit a 1-certificate with average time proportional to the product of the total conductance by the effective resistance between the root of leaves having 1-certificates [7]. If weights are independent of the input, then the learning graph is called *non-adaptive*. When they depend on previously learned bits, it is *adaptive*. But in quantum computing, we will see that they can also depend on both the value of the next queried bit and the value of the function itself! We call them *extended learning graphs*.

Formally, we generalize the original model of learning graphs by allowing two possible weights on each edge: one for positive instances and one for negative ones. Those weights are linked together as explained in the following definition.

▶ **Definition 3.1** (Extended learning graph)**.** Let $Y \subseteq Z$ be finite sets. An *extended learning graph* $\mathcal{G}$ is a 5-tuple $(\mathcal{V}, \mathcal{E}, \mathcal{S}, \{w_z^b : z \in Z, b \in \{0,1\}\}, \{p_y : y \in Y\})$ satisfying

- $(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph rooted in some vertex $r \in \mathcal{V}$;
- $\mathcal{S}$ is a vertex labelling mapping each $v \in \mathcal{V}$ to $\mathcal{S}(v) \subseteq [N]$ such that $\mathcal{S}(r) = \emptyset$ and $\mathcal{S}(v) = \mathcal{S}(u) \cup \{j\}$ for every $(u,v) \in \mathcal{E}$ and some $j \notin \mathcal{S}(u)$;
- Values $w_z^b(u,v)$ are in $\mathbb{R}_{\geq 0}$ and depend on z only through $z_{\mathcal{S}(v)}$, for every $(u,v) \in \mathcal{E}$;
- $w_x^0(u,v) = w_y^1(u,v)$ for all $x \in Z \setminus Y, y \in Y$ and edges $(u,v) \in \mathcal{E}$ such that $x_{\mathcal{S}(u)} = y_{\mathcal{S}(u)}$ and $x_j \neq y_j$ with $\mathcal{S}(v) = \mathcal{S}(u) \cup \{j\}$.
- $p_y : \mathcal{E} \to \mathbb{R}_{\geq 0}$ is a unit flow whose source is the root and such that $p_y(e) = 0$ when $w_y^1(e) = 0$, for every $y \in Y$.

We say that $\mathcal{G}$ is a *learning graph for* some function $f : Z \to \{0,1\}$, when $Y = f^{-1}(1)$ and each sink of p_y contains a 1-certificate for f on y, for all positive input $y \in f^{-1}(1)$.

We also say that $\mathcal{G}$ is an *adaptive learning graph* when $w_z^0 = w_z^1$ for all $z \in Z$. If furthermore w_z^0 is independent of z, $\mathcal{G}$ is a *non-adaptive learning graph*. Unless otherwise specified, by *learning graph* we mean *extended learning graph*.

When there is no ambiguity, we usually define $\mathcal{S}$ by stating the *label* of each vertex. We also say that an edge $e = (u,v)$ *loads* j when $\mathcal{S}(v) = \mathcal{S}(u) \cup \{j\}$. A *transition* of *length* k is a sequence of edges of the form $\big((v_1,v_2),(v_2,v_3),\ldots,(v_{k-1},v_k)\big)$, with $v_i \neq v_j$ for $i \neq j$.

The complexity of extended learning graphs is defined similarly to the one of other learning graphs by choosing the appropriate weight function for each complexity terms.

▶ **Definition 3.2** (Extended learning graph complexity)**.** Let $\mathcal{G}$ be an extended learning graph for a function $f : Z \to \{0,1\}$. Let $x \in Z \setminus f^{-1}(1)$, $y \in f^{-1}(1)$, and $\mathcal{F} \subseteq \mathcal{E}$. The *negative*

complexity of $\mathcal{F}$ on x and the *positive complexity* of $\mathcal{F}$ on y (with respect to $\mathcal{G}$) are respectively defined by

$$C^0(\mathcal{F},x)=\sum_{e\in\mathcal{F}} w_x^0(e) \quad \text{and} \quad C^1(\mathcal{F},y)=\sum_{e\in\mathcal{F}}\frac{p_y(e)^2}{w_y^1(e)}.$$

Then the *negative and positive complexities* of $\mathcal{F}$ are $C^0(\mathcal{F})=\max_{x\in f^{-1}(0)} C^0(\mathcal{F},x)$ and $C^1(\mathcal{F})=\max_{y\in f^{-1}(1)} C^1(\mathcal{F},y)$. The *complexity* of $\mathcal{F}$ is $C(\mathcal{F})=\sqrt{C^0(\mathcal{F})C^1(\mathcal{F})}$ and the *complexity* of $\mathcal{G}$ is $C(\mathcal{G})=C(\mathcal{E})$. Last, the *extended learning graph complexity* of f, denoted $\mathcal{LG}^{\text{ext}}(f)$, is the minimum complexity of an extended learning graph for f.

Most often we will split a learning graph into *stages* $\mathcal{F}$, that is, when the flow through $\mathcal{F}$ has the same total amount 1 for every positive input. This allows us to analyze the learning graph separately on each stage.

As for adaptive learning graphs [6, 8], the extended learning graph complexity is upper bounding the standard query complexity.

▶ **Theorem 3.3.** *For every function $f: Z\to\{0,1\}$, we have $Q(f)=O(\mathcal{LG}^{\text{ext}}(f))$.*

Proof. We assume that f is not constant, otherwise the result holds readily. The proof follows the lines of the analysis of the learning graph for Graph collision in [5]. We already know that $Q(f)=O(\text{Adv}^{\pm}(f))$ by Theorem 2.2. Fix any extended learning graph $\mathcal{G}$ for f. Observe from Definition 2.1 that $\text{Adv}^{\pm}(f)$ is defined by a minimization problem. Therefore finding any feasible solution with objective value $C(\mathcal{G},f)$ would conclude the proof. Without loss of generality, assume that $C^0(\mathcal{G})=C^1(\mathcal{G})$ (otherwise we can multiply all weights by $\sqrt{C^1(\mathcal{G})/C^0(\mathcal{G})}$). Then both complexities become $\sqrt{C^0(\mathcal{G})C^1(\mathcal{G})}$ and the total complexity remains $C(\mathcal{G})$.

For each edge $e=(u,v)\in\mathcal{E}$ with $\mathcal{S}(v)=\mathcal{S}(u)\cup\{j\}$, define a block-diagonal matrix $X_j^e=\sum_\alpha (Y_j^e)_\alpha$, where the sum is over all possible assignments α on $\mathcal{S}(u)$. Each $(Y_j^e)_\alpha$ is defined as $(\psi_0\psi_0^*+\psi_1\psi_1^*)$, where for each $z\in\{0,1\}^n$ and $b\in\{0,1\}$

$$\psi_b[z]=\begin{cases} p_z(e)/\sqrt{w_z^1(e)} & \text{if } z_{\mathcal{S}(u)}=\alpha,\ f(z)=1 \text{ and } z_j=1-b,\\ \sqrt{w_z^0(e)} & \text{if } z_{\mathcal{S}(u)}=\alpha,\ f(z)=0 \text{ and } z_j=b,\\ 0 & \text{otherwise.}\end{cases}$$

Define now $X_j=\sum_e X_j^e$ where the sum is over all edges e loading j. Fix any $x\in f^{-1}(0)$ and $y\in f^{-1}(1)$. Then we have $X_j^e[x,x]=w_x^0(e)$ and $X_j^e[y,y]=(p_y(e))^2/w_y^1(e)$. So the objective value is

$$\max_{z\in\{0,1\}^n}\sum_{j\in[N]} X_j[z,z]=\max\left\{\max_{x\in f^{-1}(0)}\sum_j X_j[x,x],\ \max_{y\in f^{-1}(1)}\sum_j X_j[y,y]\right\}$$
$$=\max\left\{C^0(\mathcal{G}),C^1(\mathcal{G})\right\}=C(\mathcal{G}).$$

Consider the cut $\mathcal{F}$ over $\mathcal{G}$ of edges $(u,v)\in\mathcal{E}$ such that $\mathcal{S}(v)=\mathcal{S}(u)\cup\{j\}$ and $x_{\mathcal{S}(u)}=y_{\mathcal{S}(u)}$ but $x_j\neq y_j$. Then each edge $e\in\mathcal{F}$ loading j satisfies $w_x^0(e)=w_y^1(e)$ and therefore $X_j^e[x,y]=p_y(e)$. Thus, $\sum_{j:\,x_j\neq y_j} X_j[x,y]=\sum_{e\in\mathcal{F}} p_y(e)=1$. Hence the constraints of Definition 2.1 are satisfied. ◀

3.2 Compression of learning graphs into super edges

We will simplify the presentation of our learning graphs by introducing a new type of edge encoding specific learning graphs as sub-procedures. Since an edge has a single 'exit', we can only encode learning graphs whose flows have unique sinks.

▶ **Definition 3.4** (Super edge)**.** A *super edge* e is an extended learning graph $\mathcal{G}_e$ such that each possible flow has the same unique sink. Then its *positive* and *negative edge-complexities* on input $x \in Z \setminus Y$ and $y \in Y$ are respectively $c^0(e,x) = C^0(\mathcal{G}_e, x)$ and $c^1(e,y) = C^1(\mathcal{G}_e, y)$.

Consider an edge e of a learning graph $\mathcal{G}$ with flow p. We can view e as a super edge with $c^0(e,x) = w^0_x(e)$ and $c^1(e,y) = 1/w^1_y(e)$. We have to take into account the fact that, in $\mathcal{G}$, its flow is not 1 but $p_z(e)$ for each z, so $C^0(e,x) = c^0(e,x)$ and $C^1(e,y) = p_y(e)^2 \times c^1(e,y)$. We use these notions in order to define the complexity of learning graphs with super edges.

Any learning graph with super edges is equivalent to a learning graph without super edges by doing recursively the following replacement for each super edge $e = (u_e, v_e)$: (1) replace it by its underlying learning graph $\mathcal{G}_e$, plugging the root to all incoming edges and the unique flow sink to all outgoing edges, and changing the labels as follows: we enrich the labels of vertices in $\mathcal{G}_e$ using the label of u_e, that it, if $\mathcal{S}$ and $\mathcal{S}_e$ are the vertex labelling mappings of $\mathcal{G}$ and $\mathcal{G}_e$ respectively, for each vertex w of $\mathcal{G}_e$ the label becomes $\mathcal{S}_e(w) \cup \mathcal{S}(u)$; (2) root the incoming flow as in $\mathcal{G}_e$. Let us call this learning graph the *expansion* of the original one with super edges. Then, a direct inspection leads to the following result that we will use in order to compute complexities directly on our (extended) learning graphs.

▶ **Lemma 3.5.** *Let $\mathcal{G}$ be a learning graph with super edges for some function f. Then the expansion of $\mathcal{G}$ is also a learning graph for f. Moreover, let* $\exp(\mathcal{F})$ *be the expansion of $\mathcal{F} \subseteq \mathcal{E}$. Then* $\exp(\mathcal{F})$ *has positive and negative complexities*

$$C^0(\exp(\mathcal{F}), x) = \sum_{e \in \mathcal{F}} c^0(e,x) \quad \textit{and} \quad C^1(\exp(\mathcal{F}), y) = \sum_{e \in \mathcal{F}} p_y(e)^2 \times c^1(e,y).$$

Fix some stage $\mathcal{F} \subseteq \mathcal{E}$ of $\mathcal{G}$ such that the flow through $\mathcal{F}$ has total amount 1 for each positive input. We will use the following lemma (adapted from non-adaptive learning graphs) to assume that a learning graph has positive complexity at most 1 on $\mathcal{F}$. The expectation involved here comes from the factor T, which is a parameter called the *speciality* of $\mathcal{F}$.

▶ **Lemma 3.6** (Speciality [5])**.** *Let $\mathcal{G}$ be a learning graph for $f : Z \to \{0,1\}$. Let $\mathcal{F} \subseteq \mathcal{E}$ be a stage of $\mathcal{G}$ whose flow always uses the ratio $1/T$ of transitions and every transition receives the same amount of flow. Then there is a learning graph $\widetilde{\mathcal{F}}$ composed of the edges of $\mathcal{F}$ equipped with new weights such that, denoting $c^1(e) = \max_{y' \in f^{-1}(1)} c^1(e,y')$,*

$$C^0(\widetilde{\mathcal{F}}, x) \leq T \mathop{\mathbb{E}}_{e \in \mathcal{F}} \left[c^0(e,x) c^1(e)\right] \quad \textit{and} \quad C^1(\widetilde{\mathcal{F}}, y) \leq 1, \qquad \forall x \in f^{-1}(0), y \in f^{-1}(1).$$

Proof. Let n_{total} be the number of transitions in $\mathcal{F}$ and n_{used} the number of them used by each flow (i.e. with positive flow). Therefore $T = n_{\text{total}}/n_{\text{used}}$. By assumption, the flow on each edge is either 0 or $1/n_{\text{used}}$. For each edge e in $\mathcal{F}$, let $\lambda_e = c^1(e)/n_{\text{used}}$. For every input z, we multiply $w^b_z(e)$ by λ_e, and we name $\widetilde{\mathcal{F}}$ the set $\mathcal{F}$ with the new weights. Then for any $x \in f^{-1}(0)$, $C^0(\widetilde{\mathcal{F}}, x) = \sum_{e \in \widetilde{\mathcal{F}}} \lambda_e c^0(e,x) = T \mathbb{E}_{e \in \widetilde{\mathcal{F}}} \left[c^0(e,x) c^1(e)\right]$. Similarly, for any $y \in f^{-1}(1)$, $C^1(\widetilde{\mathcal{F}}, y) = \sum_{e \in \widetilde{\mathcal{F}}} p_y(e)^2 c^1(e,y)/\lambda_e \leq 1$, since terms in the sum are positive only for edges with positive flow. ◀

3.3 Loading sparse inputs

We study a particular type of super edges, that we will use repeatedly in the sequel. To construct a learning graph for a given function, one often needs to load a subset S of the labels. This can be done by a path of length $|S|$ with negative and positive complexities $|S|$, which, after some rebalancing, leads directly to the following lemma.

▶ **Lemma 3.7.** *For any set S, there exists a super edge denoted* $\mathsf{DenseLoad}_S$ *loading S with the following complexities for any input $z \in \{0,1\}^N$:*

$$c^0(\mathsf{DenseLoad}_S, z) = |S|^2 \quad \textit{and} \quad c^1(\mathsf{DenseLoad}_S, z) = 1.$$

When the input is sparse one can do significantly better as we describe now, where $|z_S|$ denotes the Hamming weight of z_S.

▶ **Lemma 3.8.** *For any set S, there exists a super edge denoted* $\mathsf{SparseLoad}_S$ *loading S with the following complexities for any input $z \in \{0,1\}^N$:*

$$c^0(\mathsf{SparseLoad}_S, z) \leq 6|S|(|z_S|+1)\log(|S|+1) \quad \textit{and} \quad c^1(\mathsf{SparseLoad}_S, z) \leq 1.$$

Proof. Let us assume for simplicity that $N = |S|$ and $S = \{1, \dots, N\}$. We define the learning graph $\mathsf{SparseLoad}_S$ as the path through edges $e_1 = (\emptyset, \{1\})$, $e_2 = (\{1\}, \{1,2\})$, $\dots$, $e_N = (\{1, \dots, N-1\}, S)$. The weights are defined as, for $b \in \{0,1\}$ and $z \in Z$,

$$w^b_{e_j}(z) = \begin{cases} 3 \cdot (|z_{[j-1]}|+1) \cdot \log(N+1) & \text{if } z_j = b, \\ 3N \cdot \log(N+1) & \text{if } z_j = 1-b, \end{cases}$$

When $|z| > 0$, let us denote $i_0 = 0$, $i_{|z|+1} = N+1$ and $(i_k)_{k=1,\dots,|z|}$ the increasing sequence of indices j such that $z_j = 1$. Then, for $k = 1, \dots, |z|+1$, we define m_k as the number of indices $j \in (i_{k-1}, i_k)$ such that $z_j = 0$. More precisely, $m_k = i_k - i_{k-1} - 1$ for $1 \leq k \leq |z|$ and $m_{|z|+1} = N - i_{|z|}$. So $\sum_{k=1}^{|z|+1} m_k = N - |z|$. Then, for any input z,

$$C^0(\mathsf{SparseLoad}_S, z) = \begin{cases} 3N \cdot \log(N+1) & \text{if } |z| = 0, \\ 3 \cdot \left(|z|N + \sum_{i=1}^{|z|+1} i \times m_i\right) \cdot \log(N+1) & \text{otherwise,} \end{cases}$$

which is bounded above by $6N \cdot (|z|+1) \cdot \log(N+1)$. Moreover, using $\sum_{i=1}^{|z|+1} \frac{1}{i} \leq \log(|z|+1)+1$, we get

$$C^1(\mathsf{SparseLoad}_S, z) = \frac{1}{3 \cdot \log(N+1)} \left((N-|z|)\frac{1}{N} + \sum_{i=1}^{|z|+1} \frac{1}{i} \right) \leq 1. \qquad \blacktriangleleft$$

4 Composition of learning graphs

To simplify our presentation, we will use the term *empty transition* for an edge between two vertices representing the same set. They carry zero flow and weight, and they do not contribute to any complexity.

4.1 Learning graph for OR

Consider n Boolean functions $f_1, \dots, f_n$ with respective learning graphs $\mathcal{G}_1, \dots, \mathcal{G}_n$. The following lemma explains how to design a learning graph $\mathcal{G}_{\mathrm{OR}}$ for $f = \bigvee_{i \in [n]} f_i$ whose complexity is the squared mean of former ones. We will represent $\mathcal{G}_{\mathrm{OR}}$ graphically as

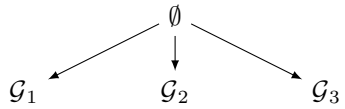
$$\emptyset \xrightarrow{\quad i \quad} \mathcal{G}_i$$

This result is similar to the one of [2], where a search procedure is designed for the case of variable query costs, or equivalently for a search problem divided into subproblems with variable complexities.

▶ **Lemma 4.1.** *Let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be learning graphs for Boolean functions $f_1, \ldots, f_n$ over Z. Assume further that for every x such that $f(x) = 1$, there is at least k functions f_i such that $f_i(x) = 1$. Then there is a learning graph $\mathcal{G}$ for $f = \bigvee_{i\in[n]} f_i$ such that for every $z \in Z$*

$$\begin{cases} C^0(\mathcal{G}, z) \leq \frac{n}{k} \times \mathop{\mathbb{E}}_{i\in[n]} \left(C^0(\mathcal{G}_i, z)C^1(\mathcal{G}_i)\right) & \text{when } f(z) = 0, \\ C^1(\mathcal{G}, z) \leq 1 & \text{when } f(z) = 1. \end{cases}$$

Proof. We define the new learning graph $\mathcal{G}$ by considering a new root $\emptyset$ that we link to the roots of each $\mathcal{G}_i$. In particular, each $\mathcal{G}_i$ lies in a different connected component. For $n = 3$, the graph is displayed below:



Then, we rescale the original weights of edges in each component $\mathcal{G}_i$ by $\lambda_i = C^1(\mathcal{G}_i)/k$. The complexity $C^0(\mathcal{G}, x)$ for a negative instance x is

$$C^0(\mathcal{G}, x) = \sum_{i=1}^{n} \lambda_i C^0(\mathcal{G}_i, x) = \frac{n}{k} \times \mathop{\mathbb{E}}_{i} \left(C^0(\mathcal{G}_i, x)C^1(\mathcal{G}_i)\right).$$

Consider now a positive instance y. Then y is also a positive instance for at least k functions f_i. Without loss of generality assume further that these k functions are $f_1, f_2, \ldots, f_k$. We define the flow of $\mathcal{G}$ (for y) as a flow uniformly directed from $\emptyset$ to $\mathcal{G}_i$ for $i = 1, 2, \ldots, k$. In each component $\mathcal{G}_i$, the flow is then routed as in $\mathcal{G}_i$. Therefore we have

$$C^1(\mathcal{G}, y) = \sum_{i=1}^{k} \frac{1}{k^2} \times \frac{C^1(\mathcal{G}_i, y)}{\lambda_i} \leq 1.$$

Finally, observe that by construction the flow is directed to sinks having 1-certificates, thus $\mathcal{G}_{\mathrm{OR}}$ indeed computes $f = \bigvee_{i\in[n]} f_i$. ◀

4.2 Learning graph for Johnson walks

We build a framework close to the one of quantum walk based algorithms from [19, 18] but for extended learning graphs. To avoid confusion we encode into a partial assignment the corresponding assigned location, that is, $z_S = \{(i, z_i) : i \in S\}$.

Fix some parameters $r \leq k \leq n$. We would like to define a learning graph $\mathcal{G}_{\mathrm{Johnson}}$ for $f = \bigvee_A f_A$, where A ranges over k-subsets of $[n]$ and f_A are Boolean functions over Z, but differently than in Lemma 4.1. For this, we are going to use a learning graph for f_A when the input has been already partially loaded, that is, loaded on $I(A)$ for some subset $I(A) \subseteq [N]$ depending on A only. Namely, we assume we are given, for every partial assignment λ, a learning graph $\mathcal{G}_{A,\lambda}$ defined over inputs $Z_\lambda = \{z \in Z : z(I(A)) = \lambda\}$ for f_A restricted to Z_λ.

Then, instead of the learning graph of Lemma 4.1, our learning graph $\mathcal{G}_{\mathrm{Johnson}}$ factorizes the load of input z over $I(A)$ for $|A| = k$ and then uses $\mathcal{G}_{A,z_{I(A)}}$. This approach is more efficient when, for every positive instance y, there is a 1-certificate $I(T_y)$ for some r-subset T_y, and $A \mapsto I(A)$ is monotone. This is indeed the analogue of a walk on the Johnson Graph.

We will represent the resulting learning graph $\mathcal{G}_{\mathrm{Johnson}}$ graphically using $r + 1$ arrows: one for the first load of $(k - r)$ elements, and r smaller ones for each of the last r loads of a single element. For example, when $r = 2$ we draw:

$$\emptyset \xrightarrow{\quad A \quad} \mathcal{G}_{A,x_{I(A)}}$$

In the following, Load_S denotes any super edge loading the elements of S, such as DenseLoad or SparseLoad that we have defined in Lemmas 3.7 and 3.8.

▶ **Theorem 4.2.** *For every subset $S \subseteq [N]$, let Load_S be any super edge loading S with $c^1(\mathsf{Load}_S) \leq 1$. Let $r \leq k \leq n$ and let $f = \bigvee_A f_A$, where A ranges over k-subsets of $[n]$ and f_A are Boolean functions over Z.*

Let I be a monotone mapping from subsets of $[n]$ to subsets of $[N]$ with the property that, for every $y \in f^{-1}(1)$, there is an r-subset $T_y \subseteq [n]$ whose image $I(T_y)$ is a 1-certificate for y.

Let $\mathbf{S}, \mathbf{U} > 0$ be such that every $x \in f^{-1}(0)$ satisfies

$$\mathop{\mathbb{E}}_{A' \subset [n] \,:\, |A'|=k-r} \left(C^0(\mathsf{Load}_{I(A')}, z)\right) \quad \leq \quad \mathbf{S}^2\,; \tag{1}$$

$$\mathop{\mathbb{E}}_{A' \subset A'' \subseteq [n] \,:\, |A'|=|A''|-1=i} \left(C^0(\mathsf{Load}_{I(A'') \setminus I(A')}, z)\right) \quad \leq \quad \mathbf{U}^2, \quad \text{for } k-r \leq i < k\,. \tag{2}$$

Let $\mathcal{G}_{A,\lambda}$ be learning graphs for functions f_A on Z restricted to inputs $Z_\lambda = \{z \in Z : z(I(A)) = \lambda\}$, for all k-subsets A of $[n]$ and all possible assignments λ over $I(A)$. Let finally $\mathbf{C} > 0$ be such that every $x \in f^{-1}(0)$ satisfies

$$\mathop{\mathbb{E}}_{A \subseteq [n] \,:\, |A|=k} \left(C^0(\mathcal{G}_{A,x_{I(A)}}, x) C^1(\mathcal{G}_{A,x_{I(A)}}, f)\right) \quad \leq \quad \mathbf{C}^2. \tag{3}$$

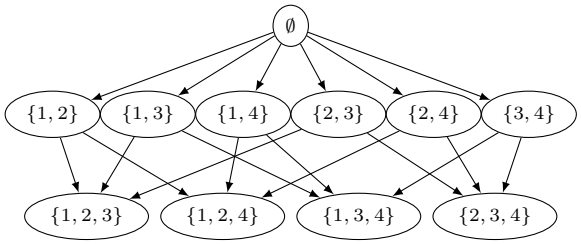
Then there is a learning graph $\mathcal{G}_{\text{Johnson}}$ for f such that for every $z \in Z$

$$\begin{cases} C^0(\mathcal{G}_{\text{Johnson}}, z) = O\left(\mathbf{S}^2 + \left(\frac{n}{k}\right)^r \left(k \times \mathbf{U}^2 + \mathbf{C}^2\right)\right) & \text{when } f(z) = 0, \\ C^1(\mathcal{G}_{\text{Johnson}}, z) = 1 & \text{when } f(z) = 1. \end{cases}$$

Proof.

Construction. We define $\mathcal{G}_{\text{Johnson}}$ by emulating a walk on the Johnson graph $J(n,k)$ for searching a k-subset A having an r-subset T_y such that $I(T_y)$ is a 1-certificate for y. In that case, by monotonicity of I, the set $I(A)$ will be also a 1-certificate for y.

Our learning graph $\mathcal{G}_{\text{Johnson}}$ is composed of $(r+2)$ stages (that is, layers whose total incoming flow is 1), that we call Stage ℓ, for $\ell = 0, 1, \ldots, r+1$. An example of such a learning graph for $n = 4$, $k = 3$ and $r = 1$ is represented below:



Stage 0 of $\mathcal{G}_{\text{Johnson}}$ consists in $\binom{n}{(k-r)}$ disjoint paths, all of same weights, leading to vertices labelled by some $(k-r)$-subset A' and loading $I(A')$. They can be implemented by the super edges $\mathsf{Load}_{I(A')}$. For positive instances y, the flow goes from $\emptyset$ to subsets $I(A')$ such that $I(A') \cap T_y = \emptyset$.

For $\ell = 1, \ldots, r$, Stage ℓ consists in $(n - (k-r) - \ell + 1)$ outgoing edges from each node labeled by a $(k - r + \ell - 1)$-subset A'. Those edges are labelled by (A', j) where $j \notin A'$ and

load $I(A' \cup \{j\}) \setminus I(A')$. They can be implemented by the super edges $\mathsf{Load}_{I(A'\cup\{j\})\setminus I(A')}$. For positive instances y, for each vertex A' getting some positive flow, the flow goes out only to the edge (A', j_ℓ), with the convention $T_y = \{j_1, \ldots, j_r\}$.

The final Stage $(r+1)$ consists in plugging in nodes A the corresponding learning graph $\mathcal{G}_{A,x_{I(A)}}$, for each k-subset A. We take a similar approach than in the construction of $\mathcal{G}_{\mathrm{OR}}$ above. The weights of the edges in each component $\mathcal{G}_{A,z_{I(A)}}$ are rescaled by a factor $\lambda_A = C^1(\mathcal{G}_{A,x_{I(A)}})/\binom{n-r}{k-r}$. For a positive instance y, the flow is directed uniformly to each $\mathcal{G}_{A,y_{I(A)}}$ such that $T_y \subseteq A$, and then according to $\mathcal{G}_{A,y_{I(A)}}$.

Observe that by construction, on positive inputs the flow reaches only 1-certificates of f. Therefore $\mathcal{G}_{\mathrm{Johnson}}$ indeed computes f.

Analysis. Remind that the positive edge-complexity of our super edge Load is at most 1.

At Stage 0, the $\binom{n}{(k-r)}$ disjoint paths are all of same weights. The flow satisfies the hypotheses of Lemma 3.6 with a speciality of $O(1)$. Therefore, using inequality (1), the complexity of this stage is $O(\mathbf{S}^2)$ when $f(x) = 0$, and at most 1 otherwise.

For $\ell = 1, \ldots, r$, at Stage ℓ consists of $(n-(k-r)-\ell+1)$ outgoing edges to each node labeled by a $(k-r+\ell-1)$-subset. Take a positive instance y. Recall that, for each vertex A' getting some positive flow, the flow goes out only to the edge (A', j_ℓ). By induction on ℓ, the incoming flow is uniform when positive. Therefore, the flow on each edge with positive flow is also uniform, and the speciality of the stage is $O((\frac{n}{k})^\ell \cdot k)$. Hence, by Lemma 3.6 and using inequality 2, the cost of each such stage is $O((\frac{n}{k})^\ell \cdot k \cdot \mathbf{U}^2)$. The dominating term is thus $O\left((\frac{n}{k})^r \cdot k \cdot \mathbf{U}^2\right)$.

The analysis of the final stage (Stage $(r+1)$) is similar to the proof of Lemma 4.1. For a negative instance x, the complexity of this stage is:

$$\begin{aligned}\sum_A \lambda_A C^0(\mathcal{G}_{A,x_{I(A)}}, x) &= \frac{\binom{n}{k}}{\binom{n-r}{k-r}} \mathop{\mathbb{E}}_A \left(C^0(\mathcal{G}_{A,x_{I(A)}}, x) C^1(\mathcal{G}_{A,x_{I(A)}})\right) \\ &= O\left(\left(\frac{n}{k}\right)^r \times \mathop{\mathbb{E}}_A \left(C^0(\mathcal{G}_{A,x_{I(A)}}, x) C^1(\mathcal{G}_{A,x_{I(A)}})\right)\right).\end{aligned}$$

Similarly, when $f(y) = 1$, we get a complexity at most 1. ◀

5 Application to Triangle Finding

5.1 An adaptive Learning graph for dense case

We start by reviewing the main ideas of Le Gall's algorithm in order to find a triangle in an input graph G with n vertices. More precisely, we decompose the problem into similar subproblems, and we build up our adaptive learning graph on top of it. Doing so, we get rid of most of the technical difficulties that arise in the resolution of the underlying problems using quantum walk based algorithms.

Let V be the vertex set of G. For a vertex u, let N_u be the neighborhood of u, and for two vertices u, v, let $N_{u,v} = N_u \cap N_v$. Figure 1 illustrates the following strategy for finding a potential triangle in some given graph G, with x, a, b integer parameters to be specified later.

First, fix an x-subset X of vertices. Then, either G has a triangle with one vertex in X or each (potential) triangle vertex is outside X. The first case is quite easy to deal with, so we ignore it for now and we only focus on the second case. Thus there is no need to query any possible edge between two vertices u, v connected to the same vertex in X. Indeed, if

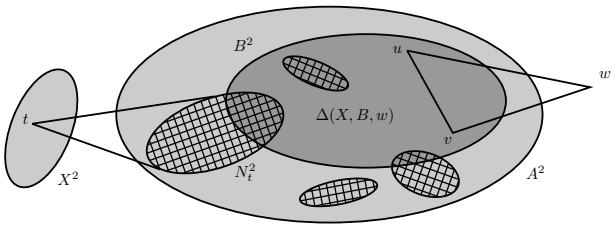


Figure 1 Sets involved in Le Gall's algorithm.

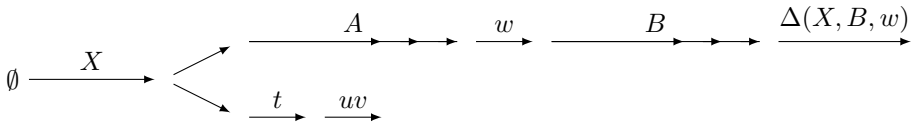


Figure 2 Learning graph for Triangle Finding with complexity $O(n^{5/4})$.

such an edge exists, the first case will detect a triangle. Therefore one only needs to look for a triangle edge in $\Delta(X) = \{(u,v) \in V^2 : N_{u,v} \cap X = \emptyset\}$.

Second, search for an a-subset A with two triangle vertices in it. For this, construct the set $\Delta(X,A) = A^2 \cap \Delta(X)$ of potential triangle edges in A^2. The set $\Delta(X,A)$ can be easily set once all edges between X and A are known.

Third, in order to decide if $\Delta(X,A)$ has a triangle edge, search for a vertex w making a triangle with an edge of $\Delta(X,A)$.

Otherwise, search for a b-subset B of A such that w makes a triangle with two vertices of B. For this last step, we construct the set $\Delta(X,B,w) = (N_w)^2 \cap \Delta(X,B)$ of pairs of vertices connected to w. If any of such pairs is an actual edge, then we have found a triangle.

We will use learning graphs of type $\mathcal{G}_{\mathrm{OR}}$ for the first step, for finding an appropriate vertex w, and for deciding weither $\Delta(X,B,w)$ has an edge; and learning graphs of type $\mathcal{G}_{\mathrm{Johnson}}$ for finding subsets A and B.

More formally now, let $\mathsf{Triangle}$ be the Boolean function such that $\mathsf{Triangle}(G) = 1$ iff graph input G has a triangle. We do the following decomposition. First, observe that $\mathsf{Triangle} = \bigvee_{X\,:\,|X|=x}(h_X \vee f_X)$ with $h_X(G) = 1$ (resp. $f_X(G) = 1$) iff G has a triangle with a vertex in X (resp. with no vertex in X). Then, we pursue the decomposition for $f_X(G)$ as $f_X(G) = \bigvee_{A\,:\,|A|=a} f_{X,A}(G)$ and $f_{X,A}(G) = \bigvee_{w\in V} f_{X,A,w}(G)$, for $A \subseteq V$ and $w \in V$, where

- $f_{X,A}(G) = 1$ iff G has a triangle between two vertices in $A \setminus X$ and a third one outside X;
- $f_{X,A,w}(G) = 1$ iff $w \notin X$ and G has a triangle between w and two vertices in $A \setminus X$.

Last, we can write $f_{X,A,w}(G) = \bigvee_{B\subset A,\ |B|=b} f_{X,B,w}(G)$.

With our notations introduced in Section 4, our adaptive learning graph $\mathcal{G}$ for Triangle Finding can be represented as in Figure 2.

Using adaptive learning graphs instead of the framework of quantum walk based algorithms from [18] simplifies the implementation of the above strategy because one can consider all the possible subsets X instead of choosing just a random one. Then one only needs to estimate the average complexity over all possible X. Such an average analysis was not considered in the framework of [18]. In addition, we do not need to estimate the size of $\Delta(X,A,w)$ at any moment of our algorithm. As a consequence, our framework greatly simplifies the combinatorial analysis of our algorithm as compared to the one of Le Gall, and lets us shave off some logarithmic factors.

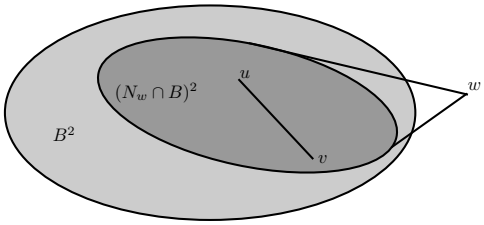


Figure 3 Sets involved in the sparse decomposition.

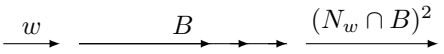


Figure 4 Learning graph for Triangle Finding with complexity $\widetilde{O}((n^{5/6}m^{1/6} + d_2\sqrt{n})\log n)$.

▶ **Theorem 5.1.** *The adaptive learning graph of Figure 2 with* $|X| = x$, $|A| = a$, $|B| = b$, *and using* Load = DenseLoad, *has complexity*

$$O\left(\sqrt{xn^2 + (ax)^2 + \left(\frac{n}{a}\right)^2\left(a\cdot x^2 + n\left(b^2 + \left(\frac{a}{b}\right)^2\left(b + \frac{b^2}{x}\right)\right)\right)}\right).$$

In particular, taking $a = n^{3/4}$ *and* $b = x = \sqrt{n}$ *leads to* $Q(\mathsf{Triangle}) = O(n^{5/4})$.

5.2 Sparse graphs

In the sparse case we now show to use extended learning graphs in order to get a better complexity than the one of Theorem 5.1.

First, the same learning graph of Theorem 5.1 has a much smaller complexity for sparse graphs when SparseLoad is used instead of DenseLoad.

▶ **Theorem 5.2.** *The learning graph of Figure 2, using* Load = SparseLoad, *has complexity over graphs with* m *edges*

$$O\left(\sqrt{\left(xm + (ax)^2\cdot\frac{m}{n^2} + \left(\frac{n}{a}\right)^2\left(a\cdot x^2\cdot\frac{m}{n^2} + n\left(b^2\cdot\frac{m}{n^2} + \left(\frac{a}{b}\right)^2\left(b + \frac{b^2}{x}\right)\right)\right)\right)\log n}\right).$$

In particular, taking $a = n^{3/4}$ *and* $b = x = \sqrt{n}/(m/n^2)^{1/3}$ *leads to a complexity of* $O(n^{11/12}m^{1/6}\sqrt{\log n})$ *when* $m \geq n^{5/4}$.

We now end with an even simpler learning graph (see Figure 3) whose complexity depends on its average of squared degrees. It consists in searching for a triangle vertex w. In order to check if w is such a vertex, we search for a b-subset B with an edge connected to w. For this purpose, we first connect w to B, and then check if there is an edge in $(N_w \cap B)^2$.

Formally, we do the decomposition $\mathsf{Triangle} = \bigvee_{w\in V} f_w$, with $f_w(G) = 1$ iff w is a triangle vertex in G. Then, we pursue the decomposition with $f_w(G) = \bigvee_{B\subseteq V\,:\,|B|=b} f_{w,B}(G)$ where $f_{w,B}(G) = 1$ iff G has a triangle formed by w and two vertices of B. Using our notations, the resulting learning graph is represented by the diagram in Figure 4.

In the following theorem, $d_2 = \sqrt{\mathbb{E}_v\left[\,|N_v|^2\,\right]}$ denotes the variance of the degrees.

▶ **Theorem 5.3.** *Let $b \geq n^2/m$. The learning graph of Figure 4, using* SparseLoad *for the first stage of $\mathcal{G}_{\text{Johnson}}$ and* DenseLoad *otherwise, has complexity over graphs with m edges*

$$O\left(\sqrt{n\left(b^2\frac{m}{n^2}\log n + \frac{n^2}{b^2}\left(b + \frac{b^2(d_2)^2}{n^2}\right)\right)}\right).$$

Taking $b = n^{4/3}/(m\log n)^{1/3}$ leads to a complexity of $O(n^{5/6}(m\log n)^{1/6} + d_2\sqrt{n})$.

References

1. S. Aaronson, S. Ben-David, and R. Kothari. Separations in query complexity using cheat sheets. In *Proceedings of 48th ACM Symposium on Theory of Computing*, pages 863–876, 2016. doi:10.1145/2897518.2897644.
2. A. Ambainis. Quantum search with variable times. *Theory of Computing Systems*, 47(3):786–807, 2010. doi:10.1007/s00224-009-9219-1.
3. A. Ambainis, K. Balodis, A. Belovs, T. Lee, M. Santha, and J. Smotrovs. Separations in query complexity based on pointer functions. In *Proceedings of 48th ACM Symposium on Theory of Computing*, pages 800–813, 2016. doi:10.1145/2897518.2897524.
4. R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, 2001.
5. A. Belovs. Learning-graph-based quantum algorithm for k-distinctness. In *Prooceedings of 53rd IEEE Symposium on Foundations of Computer Science*, pages 207–216, 2012.
6. A. Belovs. Span programs for functions with constant-sized 1-certificates. In *Proceedings of 44th Symposium on Theory of Computing Conference*, pages 77–84, 2012.
7. A. Belovs, A. Childs, S. Jeffery, R. Kothari, and F. Magniez. Time-efficient quantum walks for 3-distinctness. In *Proceedings of 40th International Colloquium on Automata, Languages and Programming*, pages 105–122, 2013.
8. A. Belovs and T. Lee. Quantum algorithm for k-distinctness with prior knowledge on the input. Technical Report arXiv:1108.3022, arXiv, 2011.
9. A. Belovs and A. Rosmanis. On the power of non-adaptive learning graphs. In *Proceedings of 28th IEEE Conference on Computational Complexity*, pages 44–55, 2013.
10. H. Buhrman, C. Dürr, M. Heiligman, P. Høyer, F. Magniez, M. Santha, and R. de Wolf. Quantum algorithms for element distinctness. *SIAM Journal on Computing*, 34(6):1324–1330, 2005.
11. F. Le Gall. Improved quantum algorithm for triangle finding via combinatorial arguments. In *Proceedings of 55th IEEE Foundations of Computer Science*, pages 216–225, 2014. doi:10.1109/FOCS.2014.31.
12. F. Le Gall and N. Shogo. Quantum algorithm for triangle finding in sparse graphs. In *Proc. of 26th International Symposium Algorithms and Computation*, pages 590–600, 2015. doi:10.1007/978-3-662-48971-0_50.
13. L. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of 28th ACM Symposium on the Theory of Computing*, pages 212–219, 1996.
14. P. Høyer, T. Lee, and R. Špalek. Negative weights make adversaries stronger. In *Proceedings of 39th ACM Symposium on Theory of Computing*, pages 526–535, 2007.
15. P. Høyer and R. Špalek. Lower bounds on quantum query complexity. *Bulletin of the European Association for Theoretical Computer Science*, 87, 2005.
16. T. Lee, F. Magniez, and M. Santha. Improved quantum query algorithms for triangle finding and associativity testing. *Algorithmica*, 2015. To appear.
17. T. Lee, R. Mittal, B. Reichardt, R. Špalek, and M. Szegedy. Quantum query complexity of state conversion. In *Proceedings of 52nd IEEE Symposium on Foundations of Computer Science*, pages 344–353, 2011.

18 F. Magniez, A. Nayak, J. Roland, and M. Santha. Search via quantum walk. *SIAM Journal on Computing*, 40(1):142–164, 2011.

19 F. Magniez, M. Santha, and M. Szegedy. Quantum algorithms for the triangle problem. *SIAM Journal on Computing*, 37(2):413–424, 2007.

20 N. Nisan. Crew prams and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, 1991. doi:10.1137/0220062.

21 B. Reichardt. Reflections for quantum query algorithms. In *Proceedings of 22nd ACM-SIAM Symposium on Discrete Algorithms*, pages 560–569, 2011.

22 P. Shor. Algorithms for quantum computation: Discrete logarithm and factoring. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.