Theoretical Computer Science 410 (2009) 5285–5297

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Claw finding algorithms using quantum walk

Seiichiro Tani *

Quantum Computation and Information Project, Solution-Oriented Research for Science and Technology, Japan Science and Technology Agency, 5-28-3 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
NTT Communication Science Laboratories, NTT Corporation, 3-1 Morinosato-Wakamiya, Atsugi, Kanagawa 243-0198, Japan

ARTICLE INFO

Keywords:
Quantum computing
Query complexity
Oracle computation
Quantum walk

ABSTRACT

The claw finding problem has been studied in terms of query complexity as one of the problems closely connected to cryptography. Given two functions, f and g, with domain sizes N and $M(N \leq M)$, respectively, and the same range, the goal of the problem is to find x and y such that $f(x) = g(y)$. This problem has been considered in both quantum and classical settings in terms of query complexity. This paper describes an optimal algorithm that uses quantum walk to solve this problem. Our algorithm can be slightly modified to solve the more general problem of finding a tuple consisting of elements in the two function domains that has a prespecified property. It can also be generalized to find a claw of k functions for any constant integer $k > 1$, where the domain sizes of the functions may be different.

1. Introduction

The most significant discoveries in quantum computation would be Shor's polynomial-time quantum algorithms for factoring integers and computing discrete logarithms [17], both of which are believed to be hard to solve in classical settings and are thus used in arguments for the security of the widely used cryptosystems. Another significant discovery is Grover's quantum algorithm for the problem of searching an unstructured set [11], i.e., the problem of searching for $i \in \{0, 1, \ldots, N-1\}$ such that $f(i) = 1$ for a hidden Boolean function f; it has yielded a variety of generalizations [3,12,2,18]. Grover's algorithm and its generalizations assume the *oracle computation model*, in which a problem instance is given as a black box (called an oracle) and any algorithm needs to make queries to the black box to get sufficient information about the instance. In the case of searching an unstructured set, any algorithm needs to make queries of the form "what is the value of function f for input i ?" to the given oracle. In the oracle computation model, the efficiency of an algorithm is usually measured by the number of queries the algorithm needs to make, i.e., by the query complexity of the algorithm. The query complexity of a problem means the query complexity of the algorithm that solves the problem with fewest queries.

One of the earliest applications of Grover's algorithm was the bounded-error algorithm of Brassard, Høyer and Tapp [4]; it addressed the *collision* problem in a cryptographic context, i.e., finding a pair (x, y) such that $f(x) = f(y)$, in a given 2-to-1 function f of domain size N. Their quantum algorithm requires $O(N^{1/3})$ queries, whereas any bounded-error classical algorithm needs $\Theta(N^{1/2})$ queries. Subsequently, Aaronson and Shi [1] proved the matching lower bound. Brassard et al. [4] considered two more related problems: the *element distinctness* problem and the *claw finding* problem. These problems are also important in a cryptographic sense. Furthermore, studying these problems has deepened our understanding of the power of quantum computation.

The element distinctness problem is to decide whether or not N integers given as an oracle are all distinct. Buhrman et al. [7] gave a bounded-error algorithm for the problem, which makes $O(N^{3/4})$ queries (strictly speaking, they assumed

* Corresponding address: NTT Communication Science Laboratories, NTT Corporation, 3-1 Morinosato-Wakamiya, Atsugi, Kanagawa 243-0198, Japan.
E-mail address: tani@theory.brl.ntt.co.jp.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.08.030

a comparison oracle, which returns just the result of comparing function values for two specified inputs, and, in this case, the query complexity is $O(N^{3/4}\log N)$). Subsequently, Ambainis [2] gave an improved upper bound $O(N^{2/3})$ by introducing a new framework of quantum walk (his quantum walk algorithm was reviewed from a slightly more general point of view in [15,9], and a much more general framework was given by Szegedy [18]). This upper bound matches the lower bound proved by Aaronson and Shi [1].

The *claw finding problem* is defined as follows. Given two functions $f : X \to Z$ and $g : Y \to Z$ as an oracle, decide whether or not there exists at least one pair $(x, y) \in X \times Y$, called a *claw*, such that $f(x) = g(y)$, and *find* a claw if it exists, where X and Y are domains of size N and M ($N \le M$), respectively. By $\mathbf{claw}_{\text{finding}}(N, M)$, we mean this problem.

After Brassard et al. [4] considered a special case of the claw finding problem, Buhrman et al. [6] gave a quantum algorithm that requires $O(N^{1/2}M^{1/4})$ queries for $N \le M < N^2$ and $O(M^{1/2})$ queries for $M \ge N^2$ (strictly speaking, they assumed a comparison oracle, and, in this case, the query complexity is multiplied by $\log N$). They also proved that any algorithm requires $\Omega(M^{1/2})$ queries by reducing the search problem over an unstructured set to the claw finding problem. Thus, while their bounds of the query complexity are tight when $M \ge N^2$, there is still a big gap when $N \le M < N^2$. They also considered the case of k functions, i.e., the *k-claw finding* problem defined as follows: given k functions $f_i : X_i := \{1, \ldots, N_i\} \to Z$ ($i \in \{1, \ldots, k\}$) as an oracle, where $k > 1$ is any constant integer, and $N_i \le N_j$ if $i < j$, decide whether or not there exists at least one k-claw, i.e., a tuple $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ such that $f_i(x_i) = f_j(x_j)$ for any $i, j \in \{1, \ldots, k\}$, and find a k-claw if it exists. A generalization of their algorithm works well for the k-claw finding problem; its query complexity is $O(N^{1-1/2^k})$ if $N_i = N$ for all $i \in \{1, \ldots, k\}$. If the promise is assumed that there is at most one solution, it has been shown in [15] that the quantum walk algorithm in [2] for the element distinctness problem is general enough to be applied with slight modification to the k-claw finding problem; this yields, with random reduction, query complexity $\tilde{O}((\sum_{i=1}^{k} N_i)^{\frac{k}{k+1}})$ for the problem without the promise of a single solution. Zhang [19] generalized the quantum walk algorithm in [2] to solve the claw finding problem with the single-solution promise by making $O((NM)^{1/3})$ queries for $N \le M < N^2$ and $O(M^{1/2})$ for $M \ge N^2$. This upper bound is optimal, since the matching lower bound $\Omega((NM)^{1/3})$ was proved in [19] by reducing the collision problem to the claw finding problem. Zhang also showed that the algorithm can be generalized to solve the more general problem of finding a tuple consisting of elements in the domains of given k functions with the single-solution promise. To solve the problems without the promise, we usually use a randomized reduction to the problem *with* the single-solution promise, which is known to increase the query complexity by at most a log factor as pointed out in [15].

This paper gives an optimal quantum algorithm that directly solves the claw finding problem without the single-solution promise. The query complexity of our algorithm is as follows:

$$Q(\mathbf{claw}_{\text{finding}}(N, M)) = \begin{cases} O\left((NM)^{1/3}\right) & (N \le M < N^2) \\ O\left(M^{1/2}\right) & (M \ge N^2), \end{cases}$$

where $Q(P)$ represents the number of queries required to solve problem P with one-sided bounded error (i.e., with the one-sided error probability bounded by a certain constant, say, 1/3). The optimality is guaranteed by the lower bounds given in [6,19]. Our algorithm can be modified to solve the more general problem of finding a tuple $(x_1, \ldots, x_p, y_1, \ldots, y_q) \in X^p \times Y^q$ such that $x_i \ne x_j$ and $y_i \ne y_j$ for any $i \ne j$, and $(f(x_1), \ldots, f(x_p), g(y_1), \ldots, g(y_q)) \in R$, for given $R \subseteq Z^{p+q}$, where p and q are positive constant integers. We call this the *(p, q)-subset finding problem* and denote it by $\mathbf{(p, q)}$-$\mathbf{subset}_{\text{finding}}(N, M)$). Thus, $\mathbf{claw}_{\text{finding}}(N, M)$ is a special case of $\mathbf{(p, q)}$-$\mathbf{subset}_{\text{finding}}(N, M)$) with $p = q = 1$ and equality relation R. The query complexity is

$$Q(\mathbf{(p, q)}\text{-}\mathbf{subset}_{\text{finding}}(N, M)) = \begin{cases} O((N^pM^q)^{1/(p+q+1)}) & N \le M < N^{1+1/q} \\ O(M^{q/(1+q)}) & M \ge N^{1+1/q}. \end{cases}$$

Our claw finding algorithm first finds subsets $\tilde{X} \subseteq X$ and $\tilde{Y} \subseteq Y$ of size $O(1)$ such that there is a claw in $\tilde{X} \times \tilde{Y}$, by using binary and 4-ary searches over X and Y; to decide with which branch we should proceed at each visited node in the search trees, we use a subroutine that *decides*, with one-sided bounded error, whether or not there exists a claw of two functions f and g. The algorithm then searches $\tilde{X} \times \tilde{Y}$ for a claw by making classical queries. If we naïvely repeat the bounded-error subroutine $O(\log M)$ times at each visited node to guarantee bounded error as a whole, a "log" factor will be multiplied to the total query complexity. Instead, at the node of depth s in the search trees, we repeat the subroutine $O(s)$ times to amplify success probability. This achieves bounded error as a whole, while pushing up the query complexity by just a constant multiplicative factor. This binary search technique can be used to solve other problems such as the search version of the element distinctness problem, together with the quantum walk in [18].

The subroutine is developed around the Szegedy's quantum walk framework [18] over a Markov chain on the graph categorical product of two Johnson graphs, which correspond to the two functions (with an idea similar to the one used in [8]). The *Johnson graph* $J(n, k)$ is a connected regular graph with $\binom{n}{k}$ vertices such that every vertex is a subset of size k of $[n]$; two vertices are adjacent if and only if the symmetric difference of their corresponding subsets has size 2. For two functions f and g with domains X and Y such that $|X| \le |Y|$, the subroutine applies Szegedy's quantum walk to the graph categorical product of two Johnson graphs $J_f = J(|X|, (|X||Y|)^{1/3})$ and $J_g = J(|Y|, (|X||Y|)^{1/3})$ if $|Y| \le |X|^2$ and to that of $J_f = J(|X|, |X|)$ and $J_g = J(|Y|, |X|)$ otherwise.

Our algorithm can be generalized to the k-claw finding problem. For the k-claw finding problem $\boldsymbol{k}$-$\mathbf{claw}_{\text{finding}}(N_1, \ldots, N_k)$ against the k functions with domain sizes N_i $(i = 1, \ldots, k)$, respectively,

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\text{finding}}(N_1, \ldots, N_k)) = \begin{cases} O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}}\right) & \text{if } \prod_{i=2}^{k} N_i = O(N_1^k), \\ O\left(\sqrt{\prod_{i=2}^{k} N_i / N_1^{k-2}}\right) & \text{otherwise.} \end{cases}$$

Our algorithms can work with slight modification even against a comparison oracle (i.e., against an oracle that, for a given pair of inputs $(x_i, x_j) \in X_i \times X_j$, only decides which is the larger of two function values $f_i(x_i)$ and $f_j(x_j)$); the query complexity increases by a multiplicative factor of $\log N_1$ for the k-function case ($\log N$ for the two-function case).

Related works

Recently, Magniez et al. [14] developed a new quantum walk over a Markov chain. One of the advantages of their quantum walk over Szegedy's is that theirs can *find* a marked vertex if there is at least one marked vertex, which would simplify our algorithm. Interestingly, our algorithm shows that Szegedy's quantum walk together with carefully adjusted binary search can find a solution to some interesting problems, such as the claw finding problem and the element distinctness problem, with the same order of query complexity.

As for the technique of gradually boosting success probability, which we use for efficient binary searches, Dürr et al. [10] used a similar idea to repeatedly search the edges of a minimum spanning tree. Høyer et al. [12] introduced an error reduction technique with a similar flavor; however, their technique is used in an algorithmic context different from ours: their error reduction is performed at each recursion level while ours is sequentially used at each step of the search tree.

Organization

Section 2 defines the problems and the oracle models considered in this paper and gives the quantum walk theorem proved in [18]. Section 3 describes algorithms that decide whether or not there are claws, (p, q)-subsets, and k-claws. These algorithms are used as subroutines in the next section. Section 4 presents algorithms for the claw finding problem, the (p, q)-subset finding problem, and the k-claw finding problem. Section 5 concludes the paper.

2. Preliminaries

This section defines problems and introduces some useful techniques. (We omit the basics of quantum computing. For reference, see standard text books, e.g., [16,13].)

We denote the set of positive integers by $\mathbb{Z}^*$, the set of $\{i : j \le i \le k \text{ for } i, j, k \in \mathbb{Z}^*\}$ by $[j.k]$, and $[1.k]$ by $[k]$ for short.

Problem 1 (*Claw Finding Problem*). Given two functions $f : X := [N] \to Z$ and $g : Y := [M] \to Z$ as an oracle for $N \le M$, where $Z := [|Z|]$, decide whether or not there exists at least one pair $(x, y) \in X \times Y$, called a claw, such that $f(x) = g(y)$, and find a claw if it exists.

We also define an easier problem as the problem of just deciding whether or not there exists at least one claw, which we call the *claw detection* problem.

In a quantum setting, the two functions are given as quantum oracle $O_{f,g}$, which is defined as $O_{f,g} : |i, h, z, w\rangle \longrightarrow |i, h, z \oplus H_i(h) \pmod{|Z|}, w\rangle$, where $i \in \{0, 1\}$ $h \in X \cup Y, z \in Z, w \in [W]$ for some $W := W(N, M) < +\infty$, $H_0(h) := f(h)$ and $H_1(h) := g(h)$. This kind of oracle, which returns the value of the function(s), is called a *standard oracle*.

Another type of oracle is called the *comparison oracle*, which, for two given inputs, only decides which is the larger of the two function values corresponding to the inputs. More formally, comparison oracle $O_{f,g}$ is defined as $O_{f,g} : |i, h_i, j, h_j, b, w\rangle \longrightarrow |i, h_i, j, h_j, b \oplus [H_i(h_i) \le H_j(h_j)], w\rangle$, where $h_i, h_j \in X \cup Y$, $i, j, b \in \{0, 1\}$, $W := W(N, M) < +\infty$, $H_0(h) := f(h)$, $H_1(h) := g(h)$, $[H_i(h_i) \le H_j(h_j)]$ is the predicate such that its value is 1 if and only if $H_i(h_i) \le H_j(h_j)$.

It is obvious that, if we are given a standard oracle, we can realize a comparison oracle by making $O(1)$ queries to the standard oracle. Thus, upper bounds for a comparison oracle are those for a standard oracle, and lower bounds for a standard oracle are those for a comparison oracle, if we ignore constant multiplicative factors.

A more general problem against the same standard oracle as given in the claw finding problem is the (p, q)-*subset finding* problem.

Problem 2 (*(p, q)-Subset Finding Problem*). Given two functions $f : X := [N] \to Z$ and $g : Y := [M] \to Z$ as a standard oracle for $N \le M$, constant positive integers p, q, and relation $R \subseteq Z^{p+q}$, (1) decide whether or not there exists at least one tuple $(x_1, \ldots, x_p, y_1, \ldots, y_q) \in X^p \times Y^q$ such that $x_i \ne x_j$ and $y_i \ne y_j$ for any $i \ne j$, and $(f(x_1), \ldots, f(x_p), g(y_1), \ldots, g(y_q)) \in R$, and (2) find such a tuple if it exists.

An easier related problem is to just decide whether or not there exists at least one tuple satisfying the above condition, which we call the (p, q)-*subset detection* problem.

Buhrman et al. [6] generalized the claw finding problem to a k-function case.

Problem 3 (*k-Claw Finding Problem*). Given k functions $f_i : X_i := [N_i] \to Z$ ($i \in [k]$) as an oracle, where $N_i \leq N_j$ if $i < j$, and $Z := [|Z|]$, decide whether or not there exists at least one k-claw, i.e., a tuple $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ such that $f_i(x_i) = f_j(x_j)$ for any $i, j \in [k]$, and find a k-claw if it exists.

An easier problem, called the *k-claw detection* problem, is defined as that of just deciding whether or not there exists at least one k-claw.

Standard and comparison oracles are defined in almost the same way as in the two-function case, except that input h belongs to one of X_i's for $i \in [k]$ and function identifier i is extended to the k-function case.

The next theorem describes Szegedy's quantum walk framework.

Theorem 1 (*[18]*). *Let $\mathcal{M}$ be a symmetric Markov chain with state set V and transition matrix P and let $\delta_{\mathcal{M}}$ be the spectral gap of P, i.e., $1 - \max_i |\lambda_i|$ for the eigenvalues λ_i's of P. For a certain subset $V' \subseteq V$ with the promise that $|V'|$ is either 0 or at least $\epsilon|V|$ for $0 < \epsilon < 1$, every element in V' is marked. For $T = O(1/\sqrt{\epsilon\delta_{\mathcal{M}}})$, the next quantum algorithm decides whether $|V'|$ is 0 ("false") or at least $\epsilon|V|$ ("true") with one-sided bounded error with cost $O(C_U + (C_F + C_W)/\sqrt{\delta_{\mathcal{M}}\epsilon})$, where $C = \sum_i |c_i\rangle\langle c_i|$ for $|c_i\rangle = \sum_j \sqrt{P_{i,j}}|i\rangle|j\rangle$ and $R = \sum_j |r_j\rangle\langle r_j|$ for $|r_j\rangle = \sum_i \sqrt{P_{j,i}}|i\rangle|j\rangle$:*

1. *Prepare $|0\rangle$ in a one-qubit register $\mathbf{R}_0$, and prepare a uniform superposition $|\phi_0\rangle := \frac{1}{\sqrt{r|V|}} \sum_{i,j\in V, P_{i,j}\neq 0} |i\rangle|j\rangle$ in a register $\mathbf{R}_1$ with cost at most C_U, where r is the number of adjacent states (of any state) in $\mathcal{M}$.*
2. *Apply the Hadamard operator $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ to $\mathbf{R}_0$.*
3. *For randomly and uniformly chosen $1 \leq t \leq T$, apply the next operation W t times to $\mathbf{R}_1$ if the content of $\mathbf{R}_0$ is "1."*
   - *3.1 To any $|i\rangle|j\rangle$, perform the next steps:* (i) *Check if $i \in V'$ with cost at most C_F.* (ii) *If $i \notin V'$, apply diffusion operator $2C - I$ with cost at most C_W.*
   - *3.2 To any $|i\rangle|j\rangle$, perform the next steps:* (i) *Check if $j \in V'$ with cost at most C_F.* (ii) *If $j \notin V'$, apply diffusion operator $2R - I$ with cost at most C_W.*
4. *Apply the Hadamard operator to $\mathbf{R}_0$, and measure registers $\mathbf{R}_0$ and $\mathbf{R}_1$ with respect to the computational basis $\{|0\rangle, |1\rangle\}$.*
5. *If the result of measuring $\mathbf{R}_0$ is 1 or a marked element is found by measuring $\mathbf{R}_1$, output "true"; otherwise output "false."*

3. Detection algorithms

In this section, we describe a "claw detection" algorithm that decides whether there exists a claw, i.e., solves the claw detection problem, and generalize the algorithm. The claw detection algorithm and its generalization will be used as subroutines in the algorithms presented in the next section.

Before presenting the claw detection algorithms, we introduce some notions. The *Johnson graph* $J(n, k)$ is a connected regular graph with $\binom{n}{k}$ vertices such that every vertex is a subset of size k of $[n]$; two vertices are adjacent if and only if the symmetric difference of their corresponding subsets has size 2. The *graph categorical product* $G := (V_G, E_G)$ of two graphs $G_1 := (V_{G_1}, E_{G_1})$ and $G_2 := (V_{G_2}, E_{G_2})$, denoted by $G := G_1 \times G_2$, is a graph having vertex set $V_G := V_{G_1} \times V_{G_2}$ such that $((v_1, v_2), (v_1', v_2')) \in E_G$ if and only if $(v_1, v_1') \in E_{G_1}$ and $(v_2, v_2') \in E_{G_2}$.

The next two propositions are useful in analyzing the claw detection algorithms we will describe.

Proposition 2. *For Markov chains $\mathcal{M}, \mathcal{M}_1, \ldots, \mathcal{M}_k$, the spectral gap δ of $\mathcal{M}$ is the minimum of those $\delta_1, \ldots, \delta_k$ of $\mathcal{M}_1, \ldots, \mathcal{M}_k$, i.e., $\delta = \min_i\{\delta_i\}$, if the underlying graph of $\mathcal{M}$ is the graph categorical product of those of $\mathcal{M}_1, \ldots, \mathcal{M}_k$.*

This is because the transition matrix of $\mathcal{M}$ is the tensor product of those matrices of $\mathcal{M}_1, \ldots, \mathcal{M}_k$.

The eigenvalues of the Markov chain on $J(n, k)$ are $\frac{(k-j)(n-k-j)-j}{k(n-k)}$ for $j \in [0.k]$ [5, pages 255–256], from which the next proposition follows.

Proposition 3. *The Markov chain on Johnson graph $J(n, k)$ has spectral gap $\delta = \Omega(1/k)$, if $2 \leq k \leq n/2$.*

3.1. Claw detection

We will first describe a claw detection algorithm against a comparison oracle, from which we can almost trivially obtain a claw detection algorithm against a standard oracle. Let Claw_Detect denote the algorithm.

3.1.1. Markov chain

To construct Claw_Detect, we apply Theorem 1 on the graph categorical product of two Johnson graphs $J_f = J(|X|, l)$ and $J_g = J(|Y|, m)$ for the domains X and Y of functions f and g, respectively, where l and m ($l \leq m$) are integers fixed later.

More precisely, let F and G be any vertices of J_f and J_g, respectively, i.e., any l-element subset and m-element subset of X and Y, respectively; (F, G) is a vertex in $J_f \times J_g$.Thus, $((F, G), (F', G'))$ is an edge connecting two vertices (F, G) and (F', G')

1 Transform $|F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle$ into $|F, G, L_{F,G}\rangle|F', G', L_{F,G}\rangle$.
2 Apply diffusion operator $2\hat{C} - I$ to obtain a superposition of $|F, G, L_{F,G}\rangle|F'', G'', L_{F,G}\rangle$ over all (F'', G'') adjacent to (F, G).
3 Transform $|F, G, L_{F,G}\rangle|F'', G'', L_{F,G}\rangle$ into $|F, G, L_{F,G}\rangle|F'', G'', L_{F'',G''}\rangle$.

Fig. 1. Implementation of the diffusion operator $2C - I$.

in $J_f \times J_g$ if and only if (F, F') and (G, G') are edges of J_f and J_g, respectively. Hereafter, $((F, G), (F', G')) \in J_f \times J_g$ means that $((F, G), (F', G'))$ is an edge of $J_f \times J_g$. We next define "marked vertices" as follows. Vertex (F, G) is marked if there is a pair of $(x, y) \in F \times G$ such that $f(x) = g(y)$. To check if (F, G) is marked or not, we just sort all elements in $F \cup G$ on their function values. Although we have to sort all elements in the initial vertex, we have only to change a small part of the sorted list we already had when moving to an adjacent vertex. This is because the sets corresponding to any two vertices that are adjacent to each other differ by only one element. For every vertex (F, G), we maintain a representation $L_{F,G}$ of the sorted list of all elements in $F \cup G$ on their function values, and we identify $(F, G, L_{F,G})$ as a vertex of $J_f \times J_g$. Here, we want to guarantee that $L_{F,G}$ is uniquely determined for any pair (F, G) in order to avoid undesirable quantum interference; we just have to introduce some appropriate rules that break ties, i.e., the situation where there are multiple elements in $F \cup G$ that have the same function value, in order to guarantee a total ordering over $X \cup Y$. To mark vertices, the algorithm checks if there exists a pair $(x, y) \in X \times Y$ such that $f(x) = g(y)$ by looking through the sorted list. Thus, another property that $L_{F,G}$ should have is to make it easy to decide whether each pair of consecutive elements in the sorted list have the same function value or not. There are many kinds of appropriate representation of $L_{F,G}$.

For instance, define a total ordering over $X \cup Y$ as follows. Let $(z_i, z_j) \in (X \cup Y) \times (X \cup Y)$. If z_i and z_j have different function values, the one having the larger function value precedes the other, which is expressed by using "$\prec$" (e.g., $z_i \prec z_j$). If z_i and z_j have the same function value, we introduce the next rule to break the tie: if (z_i, z_j) is in either $X \times Y$ or $Y \times X$, the X-element precedes the Y-element, which is expressed by using "$\preceq$"; otherwise, the element having the smaller index within X (Y) precedes the other, expressed by using "$\approx$." Then, $L_{F,G}$ has the form $z_1 \mathrm{op}_1 \cdots \mathrm{op}_{l+m-1} z_{l+m}$, where $i < j$ for $i, j \in [1.(l + m)]$ if and only if z_i precedes z_j, and $\mathrm{op}_i \in \{\prec, \preceq, \approx\}$ expresses the relation between z_i and z_{i+1} for $i \in [1.(l + m - 1)]$.

3.1.2. Quantum walk operations

As the state $|\phi_0\rangle$ in Theorem 1, we prepare

$$|\phi_0\rangle = \frac{1}{\sqrt{\binom{N}{l}\binom{M}{m} l(N-l)m(M-m)}} \bigotimes_{((F,G),(F',G'))\in J_f\times J_g} |F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle,$$

in register $\mathbf{R}_1$. To generate $|\phi_0\rangle$, we first prepare the uniform superposition of $|F, G\rangle|F', G'\rangle$ over all F, F', G, G' such that (F, F') and (G, G') are edges of J_f and J_g, respectively. Obviously, this requires no queries. We then compute $L_{F,G}$ and $L_{F',G'}$ for each basis state by making queries.

The number $1 \le t \le c/\sqrt{\delta_{\mathcal{M}}\epsilon}$ of repeating W is chosen randomly and uniformly for some constant c, and $\delta_{\mathcal{M}}$ and ϵ are fixed as follows. We set ϵ to $(l/N) \times (m/M)$, since the probability that a vertex is marked is minimized when only one claw exists for f and g, in which case the probability is $(l/N) \times (m/M)$. Since, from Proposition 3, the spectral gaps of the Markov chains on $J(N, l)$ and $J(M, m)$ are $\Omega(1/l)$ and $\Omega(1/m)$, respectively, the spectral gap $\delta_{\mathcal{M}}$ of the Markov chain $\mathcal{M}$ on $J(N, l) \times J(M, m)$ is $\Omega(\min\{1/l, 1/m\}) = \Omega(1/m)$ due to $l \le m$ and Proposition 2. Thus, $c/\sqrt{\delta_{\mathcal{M}}\epsilon} = c\sqrt{NM/l}$.

We next describe the implementation of operation W. We first check if there is a pair of $(x, y) \in F \times G$ such that $f(x) = g(y)$ by looking through $L_{F,G}$ (without any queries). For every unmarked vertex, we apply diffusion operator $2C - I$, which in our case is defined as $2\sum_{F,G} |c_{F,G}\rangle\langle c_{F,G}| - I$, where

$$|c_{F,G}\rangle := \sum_{F',G':((F,G),(F',G'))\in J_f\times J_g} \frac{1}{\sqrt{l(N-l)m(M-m)}} |F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle.$$

Since diffusion operator $2C - I$ depends on $L_{F,G}$'s, it needs to make queries to the oracle. We thus divide operator $2C - I$ into a few steps. For every unmarked vertex $(F, G, L_{F,G})$, we next transform $|F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle$ into $|F, G, L_{F,G}\rangle|F', G', L_{F,G}\rangle$ with queries to the oracle. Let $2\hat{C} - I$ be $2\sum_{F,G} |\hat{c}_{F,G}\rangle\langle\hat{c}_{F,G}| - I$, where

$$|\hat{c}_{F,G}\rangle := \sum_{F,G:((F,G),(F',G'))\in J_f\times J_g} \frac{1}{\sqrt{l(N-l)m(M-m)}} |F, G\rangle|F', G'\rangle.$$

We then perform diffusion operator $2\hat{C} - I$ on the registers where the contents "F, G" and "F', G'" are stored to obtain a superposition of $|F, G, L_{F,G}\rangle|F'', G'', L_{F,G}\rangle$ over all (F'', G'') adjacent to (F, G). Finally, we transform $|F, G, L_{F,G}\rangle|F'', G'', L_{F,G}\rangle$

Claw_Detect

Input: Integers M and N such that $M \geq N$.
Comparison oracle $O_{f,g}$ for $f : X := [N] \to Z$ and $g : Y := [M] \to Z$,
Parameters $1 \leq l \leq N$ and $1 \leq m \leq M$ such that $l \leq m$.
Output: "true" if there is a claw pair $(x, y) \in X \times Y$ such that $f(x) = f(y)$; otherwise "false".

1 Prepare $|0\rangle$ in one-qubit registers $\mathbf{R}_0$ and $\mathbf{R}_{\mathbf{mark}}$ and perform the next operations to prepare $|\phi_0\rangle$ in Theorem 1 in register $\mathbf{R}_1$.
1.1 Create a uniform superposition in $\mathbf{R}_1$,

$$|\phi_0'\rangle := \frac{1}{\sqrt{\binom{N}{l}\binom{M}{m} l(N-l)m(M-m)}} \bigotimes_{((F,G),(F',G'))\in J_f\times J_g} |F, G\rangle|F', G'\rangle.$$

1.2 Transform ϕ_0' into

$$|\phi_0\rangle := \frac{1}{\sqrt{\binom{N}{l}\binom{M}{m} l(N-l)m(M-m)}} \bigotimes_{((F,G),(F',G'))\in J_f\times J_g} |F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle,$$

with queries to the oracle.
2 Apply the Hadamard operator to $\mathbf{R}_0$.
3 Uniformly and randomly pick $1 \leq t \leq c/\sqrt{\delta_{\mathcal{M}}\epsilon} = c\sqrt{NM/l}$ for some constant c and perform the next operations t times.
3.1 To every $|F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle$, perform the next steps.
3.1.1 If there is a pair of $(x, y) \in F \times G$ such that $f(x) = g(y)$, set the content of register $\mathbf{R}_{\mathbf{mark}}$ to 1.
3.1.2 If the content of $\mathbf{R}_{\mathbf{mark}}$ is 0, apply diffusion operator $2C - I$.
3.1.3 Invert the computation in step 3.1.1 to disentangle $\mathbf{R}_{\mathbf{mark}}$.
3.2 To every $|F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle$, perform the next steps.
3.2.1 If there is a pair of $(x, y) \in F' \times G'$ such that $f(x) = g(y)$, set the content of register $\mathbf{R}_{\mathbf{mark}}$ to 1.
3.2.2 If the content of $\mathbf{R}_{\mathbf{mark}}$ is 0, apply diffusion operator $2R - I$.
3.2.3 Invert the computation in step 3.2.1 to disentangle $\mathbf{R}_{\mathbf{mark}}$.
4 Apply the Hadamard operator to $\mathbf{R}_0$ and measure registers $\mathbf{R}_0$ and $\mathbf{R}_1$ with respect to the computational basis $\{|0\rangle, |1\rangle\}$.
5 If the result of measuring $\mathbf{R}_0$ is 1 or a marked element is found by measuring $\mathbf{R}_1$, output "true"; otherwise output "false."

Fig. 2. Algorithm Claw_Detect.

into $|F, G, L_{F,G}\rangle|F'', G'', L_{F'',G''}\rangle$ with queries to the oracle. Fig. 1 summarizes $2C - I$. Diffusion operator $2R - I$ is defined as $2\sum_{F',G'} |r_{F',G'}\rangle\langle r_{F',G'}| - I$, where

$$|r_{F',G'}\rangle := \sum_{F,G:((F,G),(F',G'))\in J_f\times J_g} \frac{1}{\sqrt{l(N-l)m(M-m)}} |F, G, L_{F,G}\rangle|F', G', L_{F',G'}\rangle.$$

Operator $2R - I$ can be implemented in a way similar to $2C - I$. Fig. 2 gives a precise description of the claw detection algorithm.

Lemma 4. *Let $Q(\mathbf{claw}_{\mathrm{detect}}(N, M))$ be the number of queries needed to solve the claw detection problem for functions with domain size N and M. In the comparison oracle setting,*

$$Q(\mathbf{claw}_{\mathrm{detect}}(N, M)) = \begin{cases} O((NM)^{1/3}\log N) & (N \leq M < N^2) \\ O(M^{1/2}\log N) & (M \geq N^2). \end{cases}$$

Proof. We will estimate C_U, C_F and C_W for Claw_Detect in Fig. 2, and then apply Theorem 1.

Obviously, steps 1.1, 2, 4, and 5 need no queries. In step 1.2, each $|L_{F,G}\rangle$ and $|L_{F',G'}\rangle$ can be computed by using a reversible classical computation of sorting $(l + m)$ elements with $O((l + m)\log(l + m))$ queries to oracle $O_{f,g}$. Thus, $C_U = O((l + m)\log(l + m))$.

Steps 3.1.1 and 3.2.1 can be performed by looking through $L_{F,G}$ and $L_{F',G'}$, respectively. Since steps 3.1.3 and 3.2.3 are just the inversions of steps 3.1.1 and 3.2.1, they need no queries. Thus, $C_F = 0$.

Step 3.1.2 performs the operations described in Fig. 1. Step 2 in Fig. 1 obviously needs no queries. The other steps in Fig. 1 can be realized by inserting and deleting $O(1)$ elements to/from the sorted list of $O(l + m)$ elements. Each insertion or deletion can be performed with $O(\log(l + m))$ queries by using reversible classical computation of binary search. Thus, each run of step 3.1.2 needs $O(\log(l + m))$ queries. Similarly, each run of step 3.2.2 needs $O(\log(l + m))$ queries. Thus, we have $C_W = O(\log(l + m))$.

Since operation W (i.e., step 3) is repeated $O(\sqrt{NM/l})$ times, the total number of queries is, by Theorem 1,

$$Q(\mathbf{claw}_{\mathrm{detect}}(N,M)) = O\left((l+m)\log(l+m) + \sqrt{\frac{NM}{l}}\log(l+m)\right).$$

When $N \le M < N^2$, we set $l = m = \Theta((NM)^{1/3})$, which satisfies condition $l \le N$. The total number of queries is $O((NM)^{1/3}\log N)$. When $M \ge N^2$, we set $l = m = N$, implying that the total number of queries is $O(M^{1/2}\log N)$. □

Notice that we introduce the condition $l \le m$ to fix $\delta_{\mathcal{M}} = \min\{1/l, 1/m\}$. This is not essential; we obtain the same bound if we assume $l \ge m$.

The standard oracle case can be handled by using almost the same approach.

Corollary 5. *In the standard oracle setting,*

$$Q(\mathbf{claw}_{\mathrm{detect}}(N,M)) = \begin{cases} O((NM)^{1/3}) & (N \le M < N^2) \\ O(M^{1/2}) & (M > N^2). \end{cases}$$

Proof. In the standard oracle case, we can obtain function values by making queries; it is better to store the obtained function values for comparing them with other function values. We thus define $L_{F,G}$ in this case as a representation of the sorted list of all pairs of an element in $F \cup G$ and its function value.

The costs different from those in the case of the comparison oracle are the number of queries needed to prepare $L_{F,G}$ and $L_{F',G'}$ in step 1.2 in Fig. 2 and the number of queries needed to perform operations $2C - I$ and $2R - I$ in step 3.

Step 1.2 can compute the sorted list of $(l+m)$ pairs by obtaining their associated function values with $O(l+m)$ queries to the standard oracle. Thus, $C_U = O(l+m)$.

To realize $2C - I$, we need to perform the insertion/deletion of $O(1)$ pairs to/from an $O(l+m)$-pair sorted list. To insert/delete a pair into/from the sorted list, we only need to know its associated function value; thus, each run of step 3.1.2 needs $O(1)$ queries. Similarly, each run of step 3.2.2 needs $O(1)$ queries. Thus, we need only $C_W = O(1)$ queries.

The total number of queries is $O((l+m) + \sqrt{NM/l})$. Setting l and m in the same way as in Lemma 4 completes the proof. □

3.2. (p, q)-subset detection

The claw detection algorithm against a standard oracle can easily be modified in order to solve the problem of deciding whether there exists a tuple with prespecified property given in the (p, q)-subset finding problem.

A modification is made to the part of the algorithm that decides whether a vertex of the underlying graph is marked or not (i.e., steps 3.1.1 and 3.2.1 and their inversion, steps 3.1.3 and 3.2.3 in Fig. 2); the modification can be made without changing the number of queries. The query complexity can be analyzed by using almost the same approach as used in claw detection. When there is only one tuple with prespecified property in $X \times Y$, the number of marked vertices is $\binom{N-p}{l-p}\binom{M-q}{m-q}$. Thus, we set

$$\epsilon \ge \frac{\binom{N-p}{l-p}}{\binom{N}{l}}\frac{\binom{M-q}{m-q}}{\binom{M}{m}} \ge \frac{l^p m^q}{N^p M^q}(1 - o(1)).$$

If we assume $l \le m$, the query complexity is

$$O\left(l + m + \sqrt{\frac{mN^pM^q}{l^p m^q}}\right).$$

This function of l and m attains the minimum $O((N^pM^q)^{1/(p+q+1)})$ at $l = m = \Theta((N^pM^q)^{1/(p+q+1)})$ if $N \le M < N^{1+1/q}$, and it attains the minimum $O(M^{q/(1+q)})$ at $l = \Theta(N)$ and $m = \Theta(M^{q/(q+1)})$ if $M \ge N^{1+1/q}$ (if we assume $m \le l$, we cannot obtain better bounds).

These bounds are summarized in the next lemma.

Lemma 6. *Let $Q((\boldsymbol{p}, \boldsymbol{q})\text{-}\mathbf{subset}_{\mathrm{detect}}(N,M))$ be the number of queries needed to solve (p, q)-subset detection problem for given two functions of domain size N and M, respectively. In the standard oracle setting,*

$$Q((\boldsymbol{p}, \boldsymbol{q})\text{-}\mathbf{subset}_{\mathrm{detect}}(N,M)) = \begin{cases} O((N^pM^q)^{1/(p+q+1)}) & (N \le M < N^{1+1/q}), \\ O(M^{q/(q+1)}) & (M \ge N^{1+1/q}). \end{cases}$$

By setting $p = q = 1$, we obtain the query complexity of the claw detection problem given in Corollary 5.

3.3. k-claw detection

Our algorithm for detecting a claw can easily be generalized to the case of k functions of domains of size $N_1, \ldots, N_k$, respectively. More concretely, we apply Theorem 1 to the Markov chain on the graph categorical product of the k Johnson graphs, each of which corresponds to one of the k functions. We mean this k-claw detection algorithm by "k-Claw_Detect" in the next section.

Lemma 7. *Let $Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k))$ be the number of queries needed to solve the k-claw detection problem for any positive constant integer $k > 1$. In the comparison oracle setting,*

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k)) = \begin{cases} O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log N_1\right) & \text{if } \prod_{i=2}^{k} N_i = O(N_1^k), \\ O\left(\sqrt{\prod_{i=2}^{k} N_i / N_1^{k-2}} \log N_1\right) & \text{otherwise.} \end{cases}$$

Proof. In a way similar to that for two functions, we apply Theorem 1 on the graph categorical product of k Johnson graphs $J_{f_i} := J(|X_i|, l_i)$ $(i \in [k])$ for the domains X_i's of functions f_i's, where l_i's are integers fixed later such that $l_i \le l_j$ for $i < j$.

To create uniform superposition $|\phi_0\rangle$, we first prepare the uniform superposition of $|F_1, \ldots, F_k\rangle|F'_1, \ldots, F'_k\rangle$ over all F_i and F'_i such that (F_i, F'_i) is an edge of J_{f_i} for every i. This requires no queries. As in the two-function case, we define $L_{F_1,\ldots,F_k}$ for any $F_1, \ldots, F_k$ as a representation of the sorted list of all elements in $\bigcup_{i=1}^{k} F_i$ so that it can be uniquely determined for each tuple $(F_1, \ldots, F_k)$. We then compute $L_{F_1,\ldots,F_k}$ and $L_{F'_1,\ldots,F'_k}$ for each basis state by making $O((\sum_{i=1}^{k} l_i) \log(\sum_{i=1}^{k} l_i))$ queries to the oracle. Thus, $C_U = O((\sum_{i=1}^{k} l_i) \log(\sum_{i=1}^{k} l_i))$. C_F and C_W can be estimated as 0 and $O(\sum_{i=1}^{k} l_i)$, respectively. We set ϵ to $\prod_{i=1}^{k} l_i/N_i$ and δ to $\min_i\{1/l_i\} = 1/l_k$.

From Theorem 1, the total number of queries is

$$\begin{aligned} Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k)) &= O\left(\left(\sum_{i=1}^{k} l_i\right) \log\left(\sum_{i=1}^{k} l_i\right) + \log\left(\sum_{i=1}^{k} l_i\right) \sqrt{l_k \frac{\prod_{i=1}^{k} N_i}{\prod_{i=1}^{k} l_i}}\right) \\ &= O\left(\left(\sum_{i=1}^{k} l_i\right) \log\left(\sum_{i=1}^{k} l_i\right) + \log\left(\sum_{i=1}^{k} l_i\right) \sqrt{\frac{\prod_{i=1}^{k} N_i}{\prod_{i=1}^{k-1} l_i}}\right). \end{aligned}$$

When $\prod_{i=2}^{k} N_i = O(N_1^k)$, we set $l_i := \Theta((\prod_{i=1}^{k} N_i)^{\frac{1}{k+1}})$ for $i = 1, \ldots, k$, which satisfies condition $l_i \le N_1 \le N_i$ $(i = 1, \ldots, k)$.

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k)) = O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log\left(\prod_{i=1}^{k} N_i\right)\right) = O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log N_1\right).$$

When $\prod_{i=2}^{k} N_i = \omega(N_1^k)$, we set $l_i := \Theta(N_1)$ for $i = 1, \ldots, k$.

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k)) = O\left(\sqrt{\frac{\prod_{i=2}^{k} N_i}{N_1^{k-2}}} \log N_1\right). \quad \square$$

Notice that we introduce the condition $l_i \le l_j$ for $i < j$ to fix $\delta = \min_i\{1/l_i\}$. This is not essential; we obtain the same bound if we assume $l_{\pi[i]} \le l_{\pi[j]}$ for $i < j$ and any permutation π over $[k]$.

Against a standard oracle, we obtain the following result.

Corollary 8. *Let $Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k))$ be the number of queries needed to solve the k-claw detection problem for any positive constant integer $k > 1$. In the standard oracle setting,*

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\mathrm{detect}}(N_1, \ldots, N_k)) = \begin{cases} O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}}\right) & \text{if } \prod_{i=2}^{k} N_i = O(N_1^k), \\ O\left(\sqrt{\prod_{i=2}^{k} N_i / N_1^{k-2}}\right) & \text{otherwise.} \end{cases}$$

Claw_Search

Input: Integers N, M $(N \leq M)$.
Comparison oracle $O_{f,g}$ for $f : X := [N] \to Z$ and $g : Y := [M] \to Z$.
Output: Claw $(x, y) \in X \times Y$ such that $f(x) = g(y)$ if such a pair exists; $(-1, -1)$ otherwise.

1 Set $\tilde{X} := X$ and $\tilde{Y} := Y$.
2 Set $s := 1$, and repeat the next steps until $u_{\tilde{Y}} - l_{\tilde{Y}} \leq c|\tilde{X}|$ for some constant $c \geq 2$, where $u_{\tilde{Y}}$ and $l_{\tilde{Y}}$ are the largest and smallest values, respectively, in $\tilde{Y}$.
  2.1 Set $\Xi_Y := \{[l_{\tilde{Y}}.m_{\tilde{Y}} - 1], [m_{\tilde{Y}}.u_{\tilde{Y}}]\}$, where $m_{\tilde{Y}} := \lceil (l_{\tilde{Y}} + u_{\tilde{Y}})/2 \rceil$.
  2.2 For every $\tilde{Y}' \in \Xi_Y$, do the following.
    2.2.1 Apply Claw_Detect $(s + 2)$ times to f and g restricted to domains $\tilde{X}$ and $\tilde{Y}$, respectively.
    2.2.2 If at least one of the $(s + 2)$ results is "true," set $\tilde{Y} := \tilde{Y}'$, and go to step 2.4.
  2.3 Output $(-1, -1)$ and halt.
  2.4 Set $s := s + 1$.
3 Set $s := 1$, and repeat the next steps until $u_D - l_D \leq c$ for every $D \in \{\tilde{X}, \tilde{Y}\}$ and some constant c, say, 100, where u_D and l_D are the largest and smallest values, respectively, in D.
  3.1 For every $D \in \{\tilde{X}, \tilde{Y}\}$, set $\Xi_D := \{[l_D.u_D]\}$ if $u_D - l_D \leq c$, and
    otherwise, set $\Xi_D := \{[l_D.m_D - 1], [m_D.u_D]\}$ where $m_D := \lceil (l_D + u_D)/2 \rceil$.
  3.2 For every pair $(\tilde{X}', \tilde{Y}') \in \Xi_{\tilde{X}} \times \Xi_{\tilde{Y}}$, do the following.
    3.2.1 Apply Claw_Detect $(s + 3)$ times to f and g restricted to domains $\tilde{X}'$ and $\tilde{Y}'$, respectively.
    3.2.2 If at least one of the $(s + 3)$ results is "true," set $\tilde{X} := \tilde{X}'$ and $\tilde{Y} := \tilde{Y}'$, and go to step 3.4.
  3.3 Output $(-1, -1)$ and halt.
  3.4 Set $s := s + 1$.
4 Search $\tilde{X} \times \tilde{Y}$ for a claw by making classical queries.
5 Output claw $(x, y) \in \tilde{X} \times \tilde{Y}$ if it exists; otherwise output $(-1, -1)$.

Fig. 3. Algorithm Claw_Search.

By setting $k = 2, N_1 = N$ and $N_2 = M$ in Lemma 7 and Corollary 8, we obtain the query complexity of the claw detection problem given in Lemma 4 and Corollary 5.

4. Finding algorithms

4.1. Claw finding

We now describe an algorithm, Claw_Search, that *finds* a claw. The algorithm consists of three stages. In the first stage, we find an $O(N)$-sized subset Y' of Y such that there is a claw in $X \times Y'$, by performing binary search over Y with Claw_Detect. In the second stage, we perform 4-ary search over $X \times Y'$ with Claw_Detect to find $O(1)$-sized subsets X'' and Y'' of X and Y', respectively, such that there is a claw in $X'' \times Y''$. In the final stage, we search $X'' \times Y''$ for a claw by making classical queries. To keep the error rate moderate, say, at most 1/3, Claw_Detect is repeated $O(s)$ times against the same pair of domains at the node of depth s in the search tree at each stage. This pushes up the query complexity by only a constant multiplicative factor.

Fig. 3 precisely describes Claw_Search. Steps 2, 3, and 4 in the figure correspond to the first, second, and final stages, respectively.

Theorem 9. *In the comparison oracle setting,*

$$Q(\mathbf{claw}_{\text{finding}}(N, M)) = \begin{cases} O\left((NM)^{1/3} \log N\right) & N \leq M < N^2 \\ O(M^{1/2} \log N) & M \geq N^2. \end{cases}$$

Proof. We will analyze Claw_Search in Fig. 3.

When there is no claw, Claw_Search always outputs the correct answer. Suppose that there is a claw. The algorithm may output a wrong answer if at least one of the following two cases arises. In case (1), one of $O(\log M/N)$ runs of step 2.2 errs; in case (2), one of $O(\log N)$ runs of step 3.2 errs.

Without loss of generality, the error probability of Claw_Detect can be assumed to be at most 1/3. The error probability of each single run of step 2.2.1 is at most $1/3^{s+2}$. The error probability of each run of step 2.2 is at most $2/3^{s+2} < 1/3^{s+1}$. The error probability of case (1) is thus at most $\sum_{s=1}^{\lceil \log M/N \rceil} 1/3^{s+1} < 1/6$. The error probability of case (2) is also at most

$\sum_{s=1}^{\lceil \log N_1 \rceil} 4/3^{s+3} < \sum_{s=1}^{\lceil \log N_1 \rceil} 1/3^{s+1} < 1/6$ by a similar calculation. Therefore, the overall error probability is at most $1/6 + 1/6 = 1/3$.

We next estimate the number of queries. If $N \le M < N^2$, the size of $\tilde{Y}$ is always at most quadratically different from that of $\tilde{X}$. Thus, the sth repetition of step 2 requires $O(s(NM/2^s)^{1/3} \log N)$ queries by Lemma 4. Similarly, the sth repetition of step 3 requires $O(s(N/2^s)^{2/3} \log(N/2^s))$ queries.

The total number of queries is

$$O\left(\sum_{s=1}^{\lceil \log(M/N) \rceil} \left(s\left(N\frac{M}{2^s}\right)^{1/3} \log N\right) + \sum_{s=1}^{\lceil \log N \rceil} \left(s\left(\frac{N}{2^s}\right)^{2/3} \log \frac{N}{2^s}\right)\right) = O\left((NM)^{1/3} \log N\right).$$

If $M \ge N^2$, the sth repetition of step 2 requires $O(s((NM/2^s)^{1/3} + (M/2^s)^{1/2}) \log N)$ by Lemma 4. Thus, a similar calculation gives $O(M^{1/2} \log N)$ queries. □

We can easily obtain the standard oracle version of the above theorem by using Corollary 5 instead of Lemma 4.

Corollary 10. *In the standard oracle setting,*

$$Q(\mathbf{claw}_{\text{finding}}(N, M)) = \begin{cases} O\left((NM)^{1/3}\right) & N \le M < N^2 \\ O(M^{1/2}) & M \ge N^2. \end{cases}$$

4.2. (p, q)-subset finding

We describe an algorithm that solves the (p, q)-subset finding problem. Although the algorithm is similar to that for the claw finding problem, we have to consider that multiple elements in domain X or Y are involved with any solution (i.e., tuple) for $p > 1$ or $q > 1$.

Theorem 11. *In the standard oracle setting,*

$$Q((\boldsymbol{p}, \boldsymbol{q})\text{-}\mathbf{subset}_{finding}(N, M)) = \begin{cases} O((N^p M^q)^{1/(p+q+1)}) & N \le M < N^{1+1/q}, \\ O(M^{q/(q+1)}) & M \ge N^{1+1/q}. \end{cases}$$

Proof. As in the case of the claw finding problem, we first search for a pair of constant-sized subsets of X and Y, respectively, that contain a solution (i.e., a tuple with prespecified property) by combining r-ary search with the detection algorithm in Section 3.2.

What we need to be concerned about is that when we partition the domain into (almost-)equal-sized sub-domains, a tuple we search for may also be partitioned.

The first stage is to find a subset $\tilde{Y}_{O(N)}$ of size $O(N)$ such that there is a solution in $X \times \tilde{Y}_{O(N)}$. Suppose the following operation $\mathcal{A}$: current domain $\tilde{Y} \subseteq Y$ is randomly partitioned into two (almost-)equal-sized sub-domains $\tilde{Y}'_1$ and $\tilde{Y}'_2$, i.e., two sub-domains with size $\lceil \tilde{Y}/2 \rceil$ and $\lfloor \tilde{Y}/2 \rfloor$, followed by applying the bounded-error detection algorithm to $(X, \tilde{Y}'_1)$ and $(X, \tilde{Y}'_2)$ in order to know in which of $X \times \tilde{Y}'_1$ and $X \times \tilde{Y}'_2$ a solution exists (if there exists a solution in one of them). Operation $\mathcal{A}$ can find a random subset $\tilde{Y}' \subseteq \tilde{Y}$ with size of almost $|\tilde{Y}|/2$ such that there is a solution in $X \times \tilde{Y}'$ with at least constant probability (if there exists a solution in $X \times \tilde{Y}$) because of the following claim, which will be proved later.

Claim 1. *If there is a tuple $(x_1, \ldots, x_p, y_1, \ldots, y_q)$ with prespecified property in $X \times \tilde{Y}$, the probability $\Pr(|\tilde{Y}|)$ that $(y_1, \ldots, y_q)$ is a subset of one of $\tilde{Y}'_1$ and $\tilde{Y}'_2$ is at least some constant.*

By repeating $\mathcal{A}$ $O(1)$ times, we can find such a tuple with probability at least 2/3, i.e., with error probability at most 1/3. Let procedure $\mathcal{B}$ be these $O(1)$ repetitions of $\mathcal{A}$. We then combine $\mathcal{B}$ with r-ary search to find a subset $\tilde{Y}_{O(N)} \subseteq Y$ with error probability at most 1/6 in a way similar to the case of the claw finding problem. This is the end of the first stage.

The second stage is to find constant-sized sub-domains $\tilde{X}_{O(1)} \subseteq X$ and $\tilde{Y}_{O(1)} \subseteq \tilde{Y}_{O(N)}$ such that $\tilde{X}_{O(1)} \times \tilde{Y}_{O(1)}$ contains a solution by performing r-ary search over X and $\tilde{Y}_{O(N)}$. At the node of depth s in the search tree, the following operation is performed $O(s)$ times: current domain $\tilde{X} \subseteq X$ is randomly partitioned into (almost-)equal-sized sub-domains $\tilde{X}'_1$ and $\tilde{X}'_2$, and $\tilde{Y} \subseteq \tilde{Y}_{O(N)}$ into (almost-)equal-sized sub-domains $\tilde{Y}'_1$ and $\tilde{Y}'_2$, followed by applying the bounded-error detection algorithm to sub-domain pair $(\tilde{X}'_a, \tilde{Y}'_b)$ for every $a, b \in \{1, 2\}$. We can thus find $\tilde{X}_{O(1)}$ and $\tilde{Y}_{O(1)}$ with error probability at most 1/6 by the same argument as the first stage and the next claim:

Claim 2. *Suppose that there is a tuple $(x_1, \ldots, x_p, y_1, \ldots, y_q)$ with prespecified property in $\tilde{X} \times \tilde{Y}$. Let $\Pr(|\tilde{X}|, |\tilde{Y}|)$ be the probability that $(x_1, \ldots, x_p)$ is a subset of one of $\tilde{X}'_1$ and $\tilde{X}'_2$, and $(y_1, \ldots, y_q)$ is a subset of one of $\tilde{Y}'_1$ and $\tilde{Y}'_2$. Then, $\Pr(|\tilde{X}|, |\tilde{Y}|)$ is at least some constant.*

Therefore, the error probability is at most $1/6 + 1/6 = 1/3$ in total.

We can calculate the query complexity in a way similar to the claw finding algorithm. If $N \le M < N^{1+1/q}$, the total number of queries is, by Lemma 6,

$$O\left(\sum_{s=1}^{\lceil \log(M/N)\rceil}\left(s\left(N^p\left(\frac{M}{2^s}\right)^q\right)^{\frac{1}{p+q+1}}\right)+\sum_{s=1}^{\lceil \log N\rceil}\left(s\left(\frac{N}{2^s}\right)^{\frac{p+q}{p+q+1}}\right)\right)=O\left((N^pM^q)^{\frac{1}{p+q+1}}\right).$$

If $M \ge N^{1+1/q}$, $O(s((N^p(M/2^s)^q)^{\frac{1}{p+q+1}} + (M/2^s)^{\frac{q}{q+1}}))$ queries are made at the node of depth s in the search tree at the first stage. Thus, a similar calculation gives $O(M^{q/(q+1)})$ queries.

We now prove Claim 1. Let $S := |\tilde{Y}|$. Probability $\Pr(S)$ is at least

$$\begin{aligned}\frac{\binom{S-q}{\lceil S/2\rceil}+\binom{S}{\lceil S/2\rceil-q}}{\binom{S}{\lceil S/2\rceil}} &\ge 2\frac{\binom{S-q}{\lceil S/2\rceil}}{\binom{S}{\lceil S/2\rceil}}\\ &\ge 2\frac{(S-q)\ldots(S-q-\lceil S/2\rceil+1)}{S(S-1)\ldots(S-\lceil S/2\rceil+1)} > 2(1-2q/S))^{\lceil S/2\rceil} \ge 2e^{\frac{-(S+2)q}{S-2q}}.\end{aligned}$$

Here we use $(S-q-k)/(S-k) = 1-q/(S-k) > 1-2q/S$ for $0 \le k \le \lceil S/2\rceil - 1$ and $1-x \ge e^{\frac{-x}{1-x}}$ for $x > -1$. Since $e^{\frac{-(S+2)q}{S-2q}}$ is a monotone-increasing function of S, $\Pr(S)$ is at least $2e^{-(3q+2)}$ for $S \ge 3q$. For $S < 3q$, $\Pr(S)$ is obviously constant.

As for Claim 2, let $S_X := |\tilde{X}|$ and $S_Y := |\tilde{Y}|$. Probability $\Pr(S_X, S_Y)$ is at least

$$4\frac{\binom{S_X-p}{\lceil S_X/2\rceil}}{\binom{S_X}{\lceil S_X/2\rceil}}\frac{\binom{S_Y-q}{\lceil S_Y/2\rceil}}{\binom{S_Y}{\lceil S_Y/2\rceil}}.$$

By the same argument as in the proof of Claim 1, $\Pr(S_X, S_Y)$ is at least some constant. □

4.3. k-claw finding

Similarly, we can find a k-claw by using k-Claw_Detect as a subroutine. First, we find $O(N_1)$-sized subset X_i' of X_i for every $i \in [2.k]$ such that there is a k-claw in $X_1 \times X_2' \times \cdots \times X_k'$, by performing 2^{k-1}-ary search over X_i''s for all $i \in [2.k]$ with k-Claw_Detect. Let $X_1' := X_1$. We then perform 2^k-ary search over X_i's for all $i \in [k]$ with k-Claw_Detect to find $O(1)$-sized subset X_i'' of X_i' for every $i \in [k]$ such that there is a k-claw in $X_1'' \times \cdots \times X_k''$. Finally, we search $X_1'' \times \cdots \times X_k''$ for a k-claw by making classical queries. A more precise description of the algorithm, k-Claw_Search, is given in Fig. 4.

Theorem 12. *Let $Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\text{finding}}(N_1, \ldots, N_k))$ be the number of queries needed to solve the k-claw finding problem for any positive constant integer $k > 1$. In the comparison oracle setting,*

$$Q(\boldsymbol{k}\text{-}\mathbf{claw}_{\text{finding}}(N_1, \ldots, N_k)) = \begin{cases} O\left(\left(\prod_{i=1}^k N_i\right)^{\frac{1}{k+1}}\log N_1\right) & \text{if } \prod_{i=2}^k N_i = O(N_1^k),\\ O\left(\sqrt{\prod_{i=2}^k N_i/N_1^{k-2}}\log N_1\right) & \text{otherwise.}\end{cases}$$

Proof. When there is no claw, the algorithm always outputs the correct answer. This is because the claw detection algorithm works with one-sided error. Suppose that there is a claw. The algorithm may output a wrong answer if at least one of the following cases arises: **case (1)** one of $O(\log N_k/N_1)$ runs of step 2.2 errs; **case (2)** one of $O(\log N_1)$ runs of step 3.2 errs.

Without loss of generality, the error probability of k-Claw_Detect can be assumed to be at most 1/3. The error probability, i.e., the probability of deciding that there is no claw, of each single run of steps 2.2.1 and 2.2.2 is at most $\frac{1}{3^{(s+1)+\lceil \log_3 2^{k-1}\rceil}}$. The error probability of each run of step 2.2 is at most $\frac{2^{k-1}}{3^{(s+1)+\lceil \log_3 2^{k-1}\rceil}} \le \frac{1}{3^{s+1}}$, since the number of tuples is at most 2^{k-1} for each run of step 2.2. The error probability of case (1) is thus at most $\sum_{s=1}^{\lceil \log N_k/N_1\rceil}\frac{1}{3^{s+1}} < \frac{1}{6}$. The error probability of case (2) is also at most $\sum_{s=1}^{\lceil \log N_1\rceil}\frac{1}{3^{s+1}} < \frac{1}{6}$ by similar calculation. Therefore, the overall error probability is at most $1/6 + 1/6 = 1/3$.

The number of queries is estimated as follows. At the sth repetition of step 2, the domain of f_i has size of $O(N_1 + N_i/2^s)$. Thus, if $\prod_{i=2}^k N_i = O(N_1^k)$, the number of queries made by the sth repetition of step 2 is, by Lemma 7,

$$O\left(2^{k-1}((s+1)+\lceil \log_3 2^{k-1}\rceil)\left(\prod_{i=1}^k(N_1+N_i/2^s)\right)^{\frac{1}{k+1}}\log N_1\right)=O\left(s\left(\frac{1}{2^s}\prod_{i=1}^k N_i\right)^{\frac{1}{k+1}}\log N_1\right).$$

k-Claw_Search

Input: k integers $N_1, \ldots, N_k$ such that $N_i \le N_j$ if $i < j$.
Comparison oracle $O_{f_1,\ldots,f_k}$ for functions $f_i : X_i := [N_i] \to Z$ $(i \in [k])$.
Output: k-claw $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ such that $f_i(x_i) = f_j(x_j)$ for every $i, j \in [k]$ if it exists; $(-1, \ldots, -1)$ otherwise.

1 Set $\tilde{X}_i := X_i$ for every $i \in [k]$.
2 Set $s := 1$, and repeat the next steps until $u_i - l_i \le |\tilde{X}_1|$ for all $i \in [2.k]$, where u_i and l_i are the largest and smallest values, respectively, in $\tilde{X}_i$.
 2.1 For every $i \in [2.k]$, set $\Xi_i := \{[l_i.u_i]\}$ if $u_i - l_i \le |\tilde{X}_1|$, and
  otherwise, set $\Xi_i := \{[l_i.m_i - 1], [m_i.u_i]\}$ where $m_i := \lceil (l_i + u_i)/2 \rceil$.
 2.2 For every tuple $(\tilde{X}'_1, \tilde{X}'_2, \ldots, \tilde{X}'_k) \in \{\tilde{X}_1\} \times \Xi_2 \times \cdots \times \Xi_k$, do the following for $\beta := (s+1) + \lceil \log_3 2^{k-1} \rceil$.
  2.2.1 Apply k-Claw_Detect β times to the k functions f_i restricted to domains $\tilde{X}'_i$, respectively, for every $i \in [k]$.
  2.2.2 If at least one of the β results is "true," set $\tilde{X}_i := \tilde{X}'_i$ for every $i \in [2.k]$, and go to step 2.4.
 2.3 Output $(-1, \ldots, -1)$ and halt.
 2.4 Set $s := s + 1$.
3 Set $s := 1$, and repeat the next steps until $u_i - l_i \le c$ for all $i \in [k]$ and some constant c, say, 100, where u_i and l_i are the largest and smallest values, respectively, in $\tilde{X}_i$.
 3.1 For every $i \in [k]$, set $\Xi_i := \{[l_i.u_i]\}$ if $u_i - l_i \le c$, and
  otherwise, set $\Xi_i := \{[l_i.m_i - 1], [m_i.u_i]\}$ where $m_i = \lceil (l_i + u_i)/2 \rceil$.
 3.2 For every tuple $(\tilde{X}'_1, \tilde{X}'_2, \ldots, \tilde{X}'_k) \in \Xi_1 \times \cdots \times \Xi_k$, do the following for $\gamma := (s+1) + \lceil \log_3 2^k \rceil$.
  3.2.1 Apply k-Claw_Detect γ times to the k functions f_i restricted to domains $\tilde{X}'_i$ for every $i \in [k]$.
  3.2.2 If at least one of the γ results is "true," set $\tilde{X}_i := \tilde{X}'_i$ for every $i \in [k]$, and go to 3.4.
 3.3 Output $(-1, \ldots, -1)$ and halt.
 3.4 Set $s := s + 1$.
4 Search $\tilde{X}_1 \times \cdots \times \tilde{X}_k$ for a k-claw by making classical queries.
5 Output k-claw $(x_1, \ldots, x_k) \in X'_1 \times \cdots \times X'_k$ if it exists; otherwise output $(-1, \ldots, -1)$.

Fig. 4. Algorithm k-Claw_Search.

Similarly, the number of queries made by the sth repetition of step 3 is, by Lemma 7,

$$O\left(2^k((s+1) + \lceil \log_3 2^k \rceil)\left(\left(\frac{N_1}{2^s}\right)^{\frac{k}{k+1}} \log\left(\frac{N_1}{2^s}\right)\right)\right) = O\left(s\left(\frac{N_1^k}{2^s}\right)^{\frac{1}{k+1}} \log N_1\right).$$

The total number of queries is

$$O\left(\sum_{s=1}^{\lceil \log(N_k/N_1) \rceil} s\left(\frac{1}{2^s}\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log N_1 + \sum_{s=1}^{\lceil \log N_1 \rceil} s\left(\frac{N_1^k}{2^s}\right)^{\frac{1}{k+1}} \log N_1\right).$$

This can be simplified as $O\left(\left(\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log N_1\right)$.

If $\prod_{i=2}^{k} N_i = \omega(N_1^k)$, the number of queries made by the sth repetition of step 2 is, by Lemma 7,

$$O\left(2^{k-1}((s+1) + \lceil \log_3 2^{k-1} \rceil)\left(\sqrt{\frac{\prod_{i=2}^{k}(N_1 + N_i/2^s)}{N_1^{k-2}}} + \left(\prod_{i=1}^{k}(N_1 + N_i/2^s)\right)^{\frac{1}{k+1}}\right) \log N_1\right)$$

$$= O\left(s\sqrt{\frac{1}{2^s}\frac{\prod_{i=2}^{k} N_i}{N_1^{k-2}}} \log N_1 + s\left(\frac{1}{2^s}\prod_{i=1}^{k} N_i\right)^{\frac{1}{k+1}} \log N_1\right).$$

Thus, the total number of queries is

$$O\left(\sum_{s=1}^{\lceil \log(N_k/N_1)\rceil}\left(s\sqrt{\frac{1}{2^s}\frac{\prod_{i=2}^{k}N_i}{N_1^{k-2}}}+s\left(\frac{1}{2^s}\prod_{i=1}^{k}N_i\right)^{\frac{1}{k+1}}\right)\log N_1+\sum_{s=1}^{\lceil \log N_1\rceil}s\left(\frac{N_1^k}{2^s}\right)^{\frac{1}{k+1}}\log N_1\right).$$

This can be simplified as $O\left(\sqrt{\prod_{i=2}^{k}N_i/N_1^{k-2}}\log N_1\right)$ by using $\prod_{i=2}^{k}N_i=\omega(N_1^k)$. □

We can easily obtain the standard oracle version of the above theorem by using Corollary 8 instead of Lemma 7.

Corollary 13. *In the standard oracle setting,*

$$Q\left(\boldsymbol{k}\text{-}\mathbf{claw}_{\text{finding}}(N_1,\ldots,N_k)\right)=\begin{cases}O\left(\left(\prod_{i=1}^{k}N_i\right)^{\frac{1}{k+1}}\right) & \text{if } \prod_{i=2}^{k}N_i=O(N_1^k),\\ O\left(\sqrt{\prod_{i=2}^{k}N_i/N_1^{k-2}}\right) & \text{otherwise.}\end{cases}$$

5. Conclusion

This paper addressed an optimal quantum algorithm that solves the claw finding problem. Our algorithm uses Szegedy's quantum walk, which can directly handle the cases where there may be multiple solutions but can only decide whether there exists at least one solution. To find a solution, our algorithm combines the quantum walk with carefully adjusted classical r-ary search, which adds a constant multiplicative factor to the query complexity of the quantum walk. Our algorithm can be applied to more general problems, i.e., the (p, q)-subset finding problem and k-claw finding problem, with slight modification or generalization. The space complexity of our algorithms can be improved by decreasing the sizes of the subsets associated with vertices of the Johnson graphs that correspond to given functions. However, this increases the query complexity a lot. An open problem is how to improve the space complexity without increasing the order of the query complexity as in the case of the element distinctness problem [2].

Acknowledgments

I would like to thank several referees of MFCS'07 for their useful comments.

References

[1] S. Aaronson, Y. Shi, Quantum lower bounds for the collision and the element distinctness problems, J. ACM 51 (4) (2004) 595–605.
[2] A. Ambainis, Quantum walk algorithm for element distinctness, SIAM J. Comput. 37 (1) (2007) 21–239.
[3] G. Brassard, P. Høyer, M. Mosca, A. Tapp, Quantum amplitude amplification and estimation, in: Quantum Computation and Quantum Information: A Millennium Volume, in: AMS Contem. Math., vol. 305, American Mathematical Society, 2002, pp. 53–74.
[4] G. Brassard, P. Høyer, A. Tapp, Quantum cryptanalysis of hash and claw-free functions, in: C.L. Lucchesi, A.V. Moura (Eds.), Proceedings of the Third Latin American Symposium on Theoretical Informatics, LATIN'98, in: Lecture Notes in Computer Science, vol. 1380, Springer, 1998.
[5] A.E. Brouwer, A.M. Cohen, A. Neumaier, Distance-Regular Graphs, in: A series of Modern Surveys in Mathematics, Springer-Verlag, 1989.
[6] H. Buhrman, C. Dürr, M. Heiligman, P. Høyer, F. Magniez, M. Santha, R. de Wolf, Quantum algorithms for element distinctness, in: Proceedngs of the Sixteenth Annual IEEE Conference on Computational Complexity, 2001.
[7] H. Buhrman, C. Dürr, M. Heiligman, P. Høyer, F. Magniez, M. Santha, R. de Wolf, Quantum algorithms for element distinctness, SIAM J. Comput. 34 (6) (2005) 1324–1330.
[8] H. Buhrman, R. Špalek, Quantum verification of matrix products, in: Proceedings of the Seventeenth Annual ACM/SIAM Symposium on Discrete Algorithms SODA'06, 2006.
[9] A.M. Childs, J.M. Eisenberg, Quantum algorithms for subset finding, Quantum Inf. Comput. 5 (7) (2005) 593–604.
[10] C. Dürr, M. Heiligman, P. Høyer, M. Mhalla, Quantum query complexity of some graph problems, SIAM J. Comput. 35 (6) (2006) 1310–1328.
[11] L.K. Grover, A fast quantum mechanical algorithm for database search, in: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, 1996.
[12] P. Høyer, M. Mosca, R. de Wolf, Quantum search on bounded-error inputs, in: Proceedings of the Thirtieth International Colloquium on Automata, Languages and Programming, in: Lecture Notes in Computer Science, vol. 2719, Springer, 2003.
[13] A.Y. Kitaev, A.H. Shen, M.N. Vyalyi, Classical and Quantum Computation, in: Graduate Studies in Mathematics, vol. 47, American Mathematical Society, 2002.
[14] F. Magniez, A. Nayak, J. Roland, M. Santha, Search via quantum walk, in: D.S. Johnson, U. Feige (Eds.), Proceedings of the Thirty-Nineth Annual ACM Symposium on Theory of Computing, ACM, 2007.
[15] F. Magniez, M. Santha, M. Szegedy, Quantum algorithms for the triangle problem, SIAM J. Comput. 37 (2) (2007) 413–424.
[16] M.A. Nielsen, I.L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2000.
[17] P.W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Comput. 26 (5) (1997) 1484–1509.
[18] M. Szegedy, Quantum speed-up of markov chain based algorithms, in: Proceedings of the Forty-Fifth IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 2004.
[19] S. Zhang, Promised and distributed quantum search, in: Proceedings of the Eleventh Annual International Conference on Computing and Combinatorics, COCOON'05, in: Lecture Notes in Computer Science, vol. 3595, Springer, 2005.