2006

Classical and Quantum Algorithms for Finding Cycles

Jill Cirasella
CUNY Graduate Center

Classical and Quantum Algorithms for Finding Cycles

MSc Thesis (*Afstudeerscriptie*)

written by

Jill Cirasella

(born in 1976 in New Jersey, United States)

under the supervision of **Prof. Dr. Harry Buhrman**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
January 26, 2006	Prof. Dr. Harry Buhrman
	Dr. Benedikt Löwe
	Dr. Ronald de Wolf

Institute for Logic, Language and Computation

Abstract

Quantum computing—so weird, so wonderful—inspires much speculation about the line between the possible and the impossible. (Of course, there is still unclarity about how "impossible" intractable problems are and about how "possible" quantum computers are.) This thesis takes a slightly different tack: instead of focusing on how to make the impossible possible, it focuses on how to make the possible easier.

More specifically, this paper discusses quantum algorithms for finding cycles in graphs, a problem for which polynomial-time classical algorithms already exist. It explains and compares the classical and quantum algorithms, and it introduces a few new algorithms and observations.

However, the primary contribution of this paper is its compilation of—and coherent progression through—old and new research. Interest in quantum cycle algorithms mushroomed in 2003, and many of the new algorithms are included here. While not a comprehensive catalog, this paper is a carefully chosen selection of the most important, elegant, and efficient cycle algorithms.

Acknowledgments

Many thanks to:

- Harry Buhrman, for guidance, clarity, and patience.
- Ronald de Wolf, for insights and explanations.
- the ILLC and the Netherland-America Foundation, for the doors they opened.
- Be Birchall, Spencer Gerhardt, Tanja Hötte, Dirk Walther, and especially Julia Grodel, for creating a home away from home.
- Alexander George, for selfless (and sometimes self-endangering) support, pep talks, and gentle prods.
- my parents, for unbounded and unquestioning faith, encouragement, and love.

Contents

1 Introduction **1**

1.1 Opening Words 1

1.2 Graphs 2

1.2.1 Definitions 2

1.2.2 Representations of Graphs 3

1.2.3 Common Feature of Graph Algorithms 4

1.3 The Search Problem 4

1.3.1 Classical Search Algorithm 4

1.3.2 Quantum Search Algorithm 5

1.4 Decision Algorithms vs. Example-Finding Algorithms 5

2 Quantum Computing Basics **6**

2.1 The Very Basics 6

2.1.1 Quantum States 6

2.1.2 Quantum Operations 7

2.2 Time Complexity vs. Query Complexity 9

2.2.1 Queries 9

2.2.2 Kinds of Query Complexity 10

2.3 Quantum Algorithms 11

2.3.1 Grover's Search Algorithm 12

2.3.2 Amplitude Amplification Algorithm 14

2.3.3 Ambainis's Algorithm 15

2.3.4 Embedding Quantum Algorithms into Larger Algorithms 17

2.4 A Hypothetical Situation 17

2.5 Moving On 20

3 Triangle Algorithms **21**

3.1 Classical Triangle Algorithms 21

3.1.1 Classical Triangle Algorithm #1 21

3.1.2 Classical Triangle Algorithm #2 22

3.1.3 Classical Triangle Algorithm #3 22

3.2 Quantum Triangle Algorithms 23

3.2.1 Quantum Triangle Algorithm #1 23

3.2.2 Quantum Triangle Algorithm #2 24

3.2.3 Quantum Triangle Algorithm #3 24

3.2.4 Quantum Triangle Algorithm #4 25

3.2.5 Quantum Triangle Algorithm #5 25

3.3 Another Hypothetical Situation 29

3.4 Triangle Lower Bounds 31

4 Quadrilateral Algorithms **32**

4.1 Classical Quadrilateral Algorithms 32

4.1.1 The Obvious Adaptations 32

4.1.2 Classical Quadrilateral Algorithm #1 33

4.2 Quantum Quadrilateral Algorithms 33

4.2.1 Quantum Quadrilateral Algorithm #1 33

4.2.2 Quantum Quadrilateral Algorithm #2 34

4.3 Yet Another Hypothetical Situation 36

4.4 Quadrilateral Lower Bounds 37

5 Algorithms for Longer Cycles and Arbitrary Subgraphs **38**

5.1 Classical Algorithms for Longer Cycles 39

5.1.1 Classical Even Cycle Algorithm #1 39

5.1.2 Classical Odd Cycle Algorithm #1 41

5.1.3 Classical Even/Odd Cycle Algorithm #2 41

5.1.4 Classical Even/Odd Cycle Algorithm #3 41

5.1.5 Classical Even Cycle Algorithm #4 42

5.1.6 Even Cycle Theorem 44
5.2 Quantum Algorithms for Longer Cycles 44
5.2.1 Quantum Even Cycle Algorithm #1 45
5.3 Algorithms for Cycles of Arbitrary Length 45
5.3.1 Generalization of Ambainis's Algorithm 45
5.3.2 Generalization of $O(n^{1.3})$ Triangle Algorithm 46

6 Conclusion **47**
6.1 What Did We Do? 47
6.2 What's Left? 49

Bibliography **50**

Chapter 1

Introduction

1.1 Opening Words

Quantum computing—so weird, so wonderful—inspires much speculation about the line between the possible and the impossible. (Of course, there is still unclarity about how "impossible" intractable problems are and about how "possible" quantum computers are.) This thesis takes a slightly different tack: instead of focusing on how to make the impossible possible, it focuses on how to make the possible easier.

More specifically, this paper discusses quantum algorithms for finding cycles in graphs, a problem for which polynomial-time classical algorithms already exist. It explains and compares the classical and quantum algorithms, and it introduces a few new algorithms and observations.

However, the primary contribution of this paper is its compilation of—and coherent progression through—old and new research. Interest in quantum cycle algorithms mushroomed in 2003, and many of the new algorithms are included here. While not a comprehensive catalog, this paper is a carefully chosen selection of the most important, elegant, and efficient cycle algorithms.

The paper is organized as follows: Chapter 1 reviews graphs and the ubiquitous search problem. Chapter 2 reviews quantum computing and introduces our "bag of tricks"—three quantum procedures that appear in many quantum cycle algorithms. Chapters 3, 4, and 5 examine algorithms for finding triangles, quadrilaterals, and longer cycles, respectively. Chapter 6 summarizes the findings and presents them in a chart that facilitates comparisons.

1.2 Graphs

1.2.1 Definitions[1]

A *graph* G is a structure (V, E), where V is a finite set of *vertices* (or *nodes*) and $E \subseteq V \times V$ is a finite set of *edges*. In this paper, we examine *undirected graphs*, graphs whose edges are unordered pairs of distinct vertices. Therefore, we use (u, v) and (v, u) interchangeably, and we do not allow self-loops (u, u). A *directed graph* is a graph whose edges are ordered pairs of possibly indistinct vertices.

The size of V and the size of E are somewhat independent: $0 \leq |E| \leq \frac{|V|(|V|-1)}{2}$ in undirected graphs. Therefore, we use both $|V|$ and $|E|$ to calculate the complexities of graph algorithms. To simplify our expressions, we denote $|V|$ by n and $|E|$ by m.

Two vertices $u, v \in V$ are *adjacent* if $(u, v) \in E$. The *degree* of a vertex u is the number of vertices adjacent to it. A sequence of adjacent vertices $\langle v_0, v_1, \ldots, v_k \rangle$ is a *path* of *length* k. A path $\langle v_0, v_1, \ldots, v_k \rangle$ is a *cycle* (more specifically, a *k-cycle*) if $v_0 = v_k$ and if $v_1, \ldots, v_k$ are all distinct. We sometimes use C_k to denote a k-cycle. When k is odd, C_k is an *odd cycle*, and when k is even, C_k is an *even cycle*.

Paths and cycles are examples of subgraphs: A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Similarly, G' is a *supergraph* of G if $V \subseteq V'$ and $E \subseteq E'$.

Some graphs fall into special categories. An undirected graph $G = (V, E)$ is *complete* if every two vertices $u, v \in V$ are adjacent. And G is *bipartite* if V can be partitioned into disjoint subsets V_1 and V_2 such that every edge in E connects a vertex in V_1 to a vertex in V_2. Also, G is *dense* if $m \in \Theta(n^2)$, and G is *sparse* if $m \in o(n^2)$.

Graph problems pose questions about *nontrivial graph properties*, properties that are not true of all graphs. Such a property is connectedness: An undirected graph G is *connected* if every pair of vertices is connected by a path.

Graph properties themselves can have noteworthy properties. For example, suppose a graph G has n vertices and satisfies graph property P. Property P is *monotone* if every n-node supergraph of G also satisfies P. Having a path or cycle of a certain length is a monotone property.

---

[1]The concepts introduced in Subsections 1.2.1 and 1.2.2 are also introduced, in more or less detail, in [CLR90, pages 86–89 and 463–467] and [vL90, pages 527–537].

1.2.2 Representations of Graphs

The first step in solving a graph problem is deciding how to represent the graph G. Two data structures predominate: the adjacency matrix, which is easier to inspect for a given edge, and the adjacency list, which is easier to store. Regardless of representation, we assume that the graph's vertices are numbered $1, 2, \ldots, n$.

Adjacency Matrix

The *adjacency matrix* of a graph $G = (V, E)$ is an $n \times n$ matrix M such that $M(u, v) = 1$ if and only if $(u, v) \in E$, and $M(u, v) = 0$ otherwise. Determining whether an edge is in E takes constant time on a RAM model, but storing the matrix requires $O(n^2)$ space.

The adjacency matrix for an undirected graph is symmetric, and we need not store the redundant elements below the main diagonal. Furthermore, we can omit the main diagonal, as $M(u, u) = 0$ for all $u \in V$. However, the remaining $\frac{n(n-1)}{2}$ elements still require $O(n^2)$ storage.

Adjacency Lists

The *adjacency list representation* of a graph $G = (V, E)$ is an array A of size n. Each element of A represents a vertex $u \in V$ and points to a linked list of vertices adjacent to u.[2] Therefore, $|A[u]| = \text{degree}(u)$. An adjacency list representation requires only $O(\max(n, m)) = O(n + m)$ storage and is usually less cumbersome than the corresponding adjacency matrix. However, determining whether an edge (u, v) is in E involves searching $A[u]$ (or $A[v]$) for v (or u) and can be slow.

Once again, we can store an undirected graph more efficiently. Because the graph's vertices are numbered, the linked list for a vertex u can be limited to adjacent vertices v such that $v \geq u$. This representation halves the number of stored vertices, but it still requires $O(n + m)$ storage.

Also, the vertices v in $A[u]$ can be ordered by value. In such an *ordered list representation*, finding an edge involves a search on a sorted list, which is significantly faster than a search

[2] "Adjacency list representation" is a cumbersome term, so we often say just "adjacency list." Of course, we can also refer to the structure's n linked lists as "adjacency lists." In case of confusion, look to context for clarification.

on an unsorted list.[3] Because of this non-equivalence between ordered and unordered list representations, we should never assume that a given adjacency list is ordered.

1.2.3 Common Feature of Graph Algorithms

There are scores of graph problems and therefore scores of graph algorithms. However, most graph algorithms have a common feature: They search for something that satisfies some property. That something might be a vertex, a set of vertices, an edge, a set of edges, a subgraph, a set of subgraphs—or some combination of these.

This paper focuses on algorithms for finding cycles.[4] Almost all of the algorithms we consider include searches, and the complexity of searching affects the complexities of the algorithms. Therefore, we are keenly interested in searching, and we study it closely in the coming section and chapter.

1.3 The Search Problem

The *unordered search problem* is defined as follows: Let $N = 2^n$ for some positive integer n. Given an arbitrary bit string $x = (x_0, x_1, \ldots, x_{N-1}) \in \{0,1\}^N$, we want to find an i such that $x_i = 1$ (i.e., a *solution*) or to learn that there is no such i.[5] Note that all indices i can be represented as n-bit strings.

1.3.1 Classical Search Algorithm

Classically, the fastest way to search an unordered database is to examine each element, in order of appearance or in random order. If an element is a solution, its index is returned and the search stops. If no solution is found, the search stops after every element has been queried.

No matter what our tactics, we must examine an average of $N/2$ elements to find a unique solution. Put probabilistically, we must examine at least $N/2$ elements to find a unique

[3] A classical search on N ordered elements requires $O(\log N)$ steps, and a classical search on N unordered elements requires $O(N)$ steps (see Section 1.3).

[4] More correctly: This paper focuses on algorithms for finding and/or determining the existence of cycles. The difference between these tasks is explained in Section 1.4.

[5] Sometimes, we want to find several or all i such that $x_i = 1$.

solution with probability 1/2. So, all classical search algorithms require $O(N)$ queries.[6]

1.3.2 Quantum Search Algorithm

In 1996, Lov Grover introduced a quantum algorithm that successfully searches an unordered database in only $O(\sqrt{N})$ queries [Gro96]. Not surprisingly, Grover's algorithm is far more complicated than its classical counterpart, and Chapter 2 is devoted to explaining both the algorithm and the quantum computational basics required to understand it.

1.4 Decision Algorithms vs. Example-Finding Algorithms

When we ask a question, we sometimes want a yes/no answer and sometimes want a detailed answer. And when we want a yes/no answer, we rarely get just a yes or just a no. Instead, we get arguments and evidence—from which we can deduce a yes or a no. Similarly, many computer problems are yes/no questions, and many algorithms for these problems do more than return yes or no.

Computational yes/no problems are known as *decision problems*. A common decision problem is: "Does input X contain a structure Y?" Often, the fastest and simplest way to answer the question is to look for an example of the desired structure. Furthermore, an algorithm that supplies an example is usually more useful than an algorithm that does not. However, a correct yes/no answer is still a fully correct answer to a decision problem, and we are equally interested in decision algorithms and example-finding algorithms. That said, it is sometimes instructive to transform a decision algorithm into an example-finding algorithm. These transformations are called *search-to-decision reductions*, and we will study several of them in the coming chapters.

[6] From this point forward, we are more interested in query complexity than time complexity. See Section 2.2 for an explanation of query complexity.

Chapter 2

Quantum Computing Basics

In this chapter, we introduce three quantum procedures that appear over and over again in quantum graph algorithms: Grover's quantum search algorithm, amplitude amplification, and Ambainis' algorithm. Like all quantum algorithms, these procedures exploit the weirdness of quantum physics, and we can understand them only if we are familiar with some quantum basics. Therefore, Section 2.1 summarizes the very basics of quantum mechanics, and Section 2.2 introduces the basics of quantum query complexity. For more thorough treatments, see [NC00] and [dW01].

2.1 The Very Basics

Let us say that *classical computing* involves transitions between classical states and that *quantum computing* involves transitions between quantum states. These definitions are simplistic, but they make stark the fact that understanding quantum computing amounts to understanding quantum states and the acceptable transitions between them.

2.1.1 Quantum States

As is well known, a classical state φ consists of one or more classical bits. Likewise, a quantum state $|\varphi\rangle$ consists of one or more quantum bits, called *qubits*. However, a classical bit can be only 0 or 1, while a qubit can be $|0\rangle$, $|1\rangle$, or a superposition of both $|0\rangle$ and $|1\rangle$. (States $|0\rangle$ and $|1\rangle$ are quantum basis states and correspond to the vectors $\binom{1}{0}$ and $\binom{0}{1}$, respectively.) The complex amplitudes α_0 of $|0\rangle$ and α_1 of $|1\rangle$ define the state of a qubit, and when both α_0 and α_1 are nonzero, the qubit is in a superposition. However, a superposition cannot be observed.

Observation jostles a qubit out of its superposition and projects, or collapses, it onto either $|0\rangle$ or $|1\rangle$. The probability of observing $|0\rangle$ is $|\alpha_0|^2$, and the probability of observing $|1\rangle$ is $|\alpha_1|^2$, so $|\alpha_0|^2 + |\alpha_1|^2$ must equal 1.

The importance of superposed states becomes more obvious when we consider that, while an n-bit classical register can hold any one number between 0 and $2^n - 1$, an n-qubit quantum register can hold in superposition *every* number between 0 and $2^n - 1$. Moreover, computations made on such a quantum register are made simultaneously on all 2^n numbers.

An n-qubit quantum state $|\varphi\rangle$ is written:

$$\alpha_0|0\rangle + \alpha_1|1\rangle + \cdots + \alpha_{2^n-1}|2^n - 1\rangle = \sum_{i=0}^{2^n-1} \alpha_i|i\rangle,$$

where $\alpha_0, \alpha_1, \ldots, \alpha_{2^n-1}$ are complex amplitudes and $|0\rangle, |1\rangle, \ldots, |2^n - 1\rangle$ are quantum basis states. Note that $|\varphi\rangle$ can also be written in vector notation:

$$\alpha_0 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \alpha_1 \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \cdots + \alpha_{2^n-1} \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{2^n-1} \end{pmatrix}.$$

It is always the case that $|\alpha_0|^2 + |\alpha_1|^2 + \cdots + |\alpha_{2^n-1}|^2 = 1$, which is equivalent to saying that $|\varphi\rangle$ always has norm 1. Also, observation always projects $|\varphi\rangle$ to state $|i\rangle$ with probability $|\alpha_i|^2$.

2.1.2 Quantum Operations

All quantum operations, like all classical operations, can be expressed as matrices. However, only some matrices define quantum operations. Specifically, a matrix U defines a realizable quantum operation if and only if it is *unitary* (that is, if and only if it is a square matrix whose inverse U^{-1} equals its conjugate transpose U^*). Unitary matrices preserve the norms of vectors, so, no matter how many quantum operations are applied to a quantum state, the resulting state always has norm 1. Because we can write quantum operations as matrices and quantum states as vectors, we can calculate the effect of a quantum operation on a quantum state by multiplying the operation's matrix U by the state's vector $|\varphi\rangle$: $U|\varphi\rangle = |\varphi'\rangle$.

A very important quantum operation is the *Hadamard transform*:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Applying H to $|0\rangle$ yields $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and applying H to $|1\rangle$ yields $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Because $|\pm\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$, both of these superpositions have a 50% chance of projecting onto $|0\rangle$ and a 50% chance of projecting onto $|1\rangle$. In other words, H puts a qubit into a uniform superposition of its basis states.

If we do not observe these uniform superpositions but again apply H, our original states reappear. Observe:

$$H\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \left(\frac{H|0\rangle + H|1\rangle}{\sqrt{2}}\right) = \left(\frac{\frac{|0\rangle+|1\rangle}{\sqrt{2}} + \frac{|0\rangle-|1\rangle}{\sqrt{2}}}{\sqrt{2}}\right) = \frac{|0\rangle}{2} + \frac{|1\rangle}{2} + \frac{|0\rangle}{2} - \frac{|1\rangle}{2} = |0\rangle;$$

$$H\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) = \left(\frac{H|0\rangle - H|1\rangle}{\sqrt{2}}\right) = \left(\frac{\frac{|0\rangle+|1\rangle}{\sqrt{2}} - \frac{|0\rangle-|1\rangle}{\sqrt{2}}}{\sqrt{2}}\right) = \frac{|0\rangle}{2} + \frac{|1\rangle}{2} - \frac{|0\rangle}{2} + \frac{|1\rangle}{2} = |1\rangle.$$

These second applications of H allow quantum interference to appear. Let us examine what happens to $|0\rangle$ (the behavior of $|1\rangle$ is analogous): After one application of H to $|0\rangle$, the amplitudes of $|0\rangle$ and $|1\rangle$ are equal. But, after a second application of H, the amplitude of $|0\rangle$ is $\frac{1}{2} + \frac{1}{2} = 1$, and the amplitude of $|1\rangle$ is $\frac{1}{2} - \frac{1}{2} = 0$. The amplitudes of $|0\rangle$ interfere constructively, the amplitudes of $|1\rangle$ interfere destructively, and we always observe $|0\rangle$.

The Hadamard transform is a one-qubit operation, but it can be simultaneously applied to multiple qubits. For example, n Hadamard transforms can be used to put a register of n zeroes in uniform superposition:[1]

$$H \otimes H \otimes \cdots \otimes H|0^n\rangle = H^{\otimes n}|0^n\rangle = H|0\rangle \otimes H|0\rangle \otimes \cdots \otimes H|0\rangle =$$
$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) \otimes \cdots \otimes \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) =$$
$$\frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} |j\rangle.$$

Of course, $H^{\otimes n}$ can be applied to any n-qubit basis state, not just $|0^n\rangle$. Applying $H^{\otimes n}$ to an arbitrary n-qubit state $|i\rangle$ yields:

$$H^{\otimes n}|i\rangle = \frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} (-1)^{i \cdot j}|j\rangle,$$

[1]The following equations introduce the tensor product notation $\otimes$. *Tensor product* is a way of combining vector spaces or matrices (in this context, quantum states or quantum operations, respectively). The tensor product of states $|u\rangle$ and $|v\rangle$ is the linear combination of $|u\rangle$ and $|v\rangle$ and can be written $|u\rangle \otimes |v\rangle$, $|u\rangle|v\rangle$, $|u, v\rangle$, or $|uv\rangle$. The tensor product of operations A and B is the operation that simultaneously applies A to some bits and B to other bits. So, if A is an m-bit operation, B is an n-bit operation, $|u\rangle$ is an m-bit state, and $|v\rangle$ is an n-bit state, then: $A \otimes B(|u\rangle \otimes |v\rangle) = A|u\rangle \otimes B|v\rangle$.

where $i \cdot j$ is the inner product of i and j. As we will see later, $H^{\otimes n}$ is a common operation in many quantum algorithms.

2.2 Time Complexity vs. Query Complexity

When we evaluate a classical algorithm, we are very concerned with its time consumption. Therefore, we compute its *time complexity*, the number of operations it makes when running on its worst-case input. We can also compute a quantum algorithm's time complexity: Each gate counts as one operation. So, whenever we know a quantum algorithm's sequence of gates, we can compare its time complexity to the time complexity of its fastest classical counterpart.

However, for many problems, we do not know whether the fastest known classical algorithm is also the fastest possible classical algorithm. In other words, we do not know the lower bound on the problem's classical time complexity. As a result, we cannot know whether a quantum algorithm's time complexity is better than the problem's classical lower bound—which means that we cannot fully know how much more powerful quantum computing is than classical computing.[2]

Therefore, an algorithm's time complexity is sometimes less useful than its *query complexity*, the number of times the algorithm accesses its worst-case input. The query complexity model [BBC+98] is simple, and an algorithm's query complexity says less about its efficiency than its time complexity does. However, when we do not know the lower bound on a problem's classical time complexity, we sometimes do know the lower bound on its classical query complexity. In such cases, we can learn more by comparing quantum and classical query complexities than by comparing quantum and classical time complexities.

2.2.1 Queries[3]

Suppose that an algorithm takes an input $x = (x_1, x_2, \ldots, x_N) \in \{0,1\}^N$, where $N = 2^n$ for some positive integer n. Suppose also that we can ask a oracle to return the value of any bit x_i. Each such request is called a *query*. Once a bit is queried, its value is known, and it

[2]Put in terms of complexity classes: If we had a polynomial-time quantum algorithm for a problem known to require exponential time by classical probabilistic algorithms, we would know that $BPP \subset BQP$. For more on complexity classes, see [dW01] and [Pap94].

[3]Subsections 2.2.1 and 2.2.2 summarize [dW01, Sections 1.5 and 2.3].

never needs to be queried again. Therefore, the query complexity of any algorithm, classical or quantum, is bounded above by N.

In a quantum algorithm, the following unitary transformation can act as an oracle:

$$O : |i\rangle|b\rangle \rightarrow |i\rangle|b \oplus x_i\rangle,$$

where $i \in \{0,1\}^n$, $b \in \{0,1\}$, and $\oplus$ is addition modulo 2. Because O is a quantum transformation, we can apply it to a superposition of several index values i, in which case the resulting $|b \oplus x_i\rangle$ is also a superposition. Each application of O counts as one query (even when it is applied to a superposition).

If we set b to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and apply O to $|i\rangle \left[\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right]$, then we get:

$$|i\rangle \left[\frac{1}{\sqrt{2}}\left(|0 \oplus x_i\rangle - |1 \oplus x_i\rangle\right)\right] = |i\rangle \left[\frac{1}{\sqrt{2}}\left(|x_i\rangle - |1 - x_i\rangle\right)\right] = (-1)^{x_i}|i\rangle \left[\frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right)\right].$$

We can ignore the second qubit and use $O_\pm$ to denote the transformation from $|i\rangle$ to $(-1)^{x_i}|i\rangle$. Note that $O_\pm$ flips the input state's phase if $x_i = 1$ and preserves its phase if $x_i = 0$. As with O, each application of $O_\pm$ counts as one query.

An algorithm might perform many operations in addition to its queries, and its time complexity might be much greater than its query complexity. Therefore, a quantum algorithm and a classical algorithm with very different query complexities might not have similarly separated time complexities. In other words, query complexities imply nothing about the relationships between P, NP, and the other time complexity classes.

2.2.2 Kinds of Query Complexity

Below are six kinds of algorithms for computing a function f. For each kind of algorithm, we use a different query complexity notation:

$D(f)$: The query complexity of a *deterministic algorithm*, a classical algorithm that is always correct and has no randomness.

$R_2(f)$: The query complexity of a *bounded-error randomized algorithm*, a classical algorithm that employs coin flips and is correct with probability at least 1/2. (Often, the threshold probability is defined to be 2/3. The exact threshold is not important: As long as the threshold is at least 1/2, the success probability can be boosted arbitrarily high with repetitions.)

$R_0(f)$: The query complexity of a *zero-error randomized algorithm*, a classical algorithm that employs coin flips and is never incorrect but is permitted to terminate without an answer with probability at most 1/2.

$Q_E(f)$: The query complexity of a *exact quantum algorithm*, a quantum algorithm that is always correct.

$Q_2(f)$: The query complexity of a *bounded-error quantum algorithm*, a quantum algorithm that is correct with probability at least 1/2 (or 2/3, or another constant value $\geq 1/2$).

$Q_0(f)$: The query complexity of a *zero-error quantum algorithm*, a quantum algorithm that is never incorrect but is permitted to terminate without an answer with probability at most 1/2.

We do not know all of the relationships between the query complexities, but we do know that $Q_2(f) \leq R_2(f) \leq R_0(f) \leq D(f) \leq N$ and that $Q_2(f) \leq Q_0(f) \leq Q_E(f) \leq D(f) \leq N$.

2.3 Quantum Algorithms

The first few quantum algorithms were devised by David Deutsch and Richard Jozsa, Ethan Bernstein and Umesh Vazirani, and Daniel Simon ([DJ92], [BV97], and [Sim97], respectively). Their algorithms require significantly fewer queries than the best possible classical algorithms for their problems. However, the problems are rather contrived, and their algorithms were overshadowed by Peter Shor's polynomial-time algorithms for finding prime factors and discrete logarithms [Sho97]. Shor's algorithms are expontentially faster than the fastest known classical algorithms, and they generated massive excitement about the power of quantum computing.[4]

The other most important early quantum algorithm is Lov Grover's quantum search algorithm [Gro96], which is quadratically faster than the best possible classical algorithm. Grover's algorithm is notable because it was the first quantum algorithm for a *total problem*,

[4]It is still unknown how much time is required to find prime factors and discrete logarithms classically. But, if it is ever proven that exponential time is required, then we will know that quantum computing is significantly more powerful than classical computing, and we will have a first counterexample to the strong Church's thesis [Pap94, page 36].

a problem whose input can be any of the 2^N possible inputs of length N.[5] Grover's algorithm is explained in detail below, as are a generalization of Grover's algorithm and Andris Ambainis's recent algorithm. These three algorithms become our "bag of tricks" for creating other quantum algorithms.

2.3.1 Grover's Search Algorithm[6]

Grover's search algorithm solves the unordered search problem defined in Section 1.3. Suppose for now that the input contains exactly one solution. (We will consider inputs with no solutions and multiple solutions later.)

The heart of Grover's algorithm is a sequence of gates known as the *Grover iterate*. The Grover iterate is $G = (-I + [2/N])O_x$, where I is the $N \times N$ identity matrix, $[2/N]$ is the $N \times N$ matrix with $2/N$ in every entry, and O_x is the oracle $O_\pm$ for the input x. The unitary matrix $(-I + [2/N])$ is called the *diffusion transform* and can be implemented as $-H^{\otimes n} O_G H^{\otimes n}$, where $O_G|\vec{0}\rangle = -|\vec{0}\rangle$ and does nothing to all other states.

The Grover iterate is embedded as follows:

Grover's Search Algorithm [Gro96]

1. Begin in the n-qubit state $|\vec{0}\rangle$.
2. Apply the Hadamard transform H to every qubit in $|\vec{0}\rangle$, resulting in the uniform superposition $\frac{1}{\sqrt{N}} \sum_i |i\rangle$.
3. Apply the Grover iterate $O(\sqrt{N})$ times.
4. Measure the resulting superposition, collapsing it into a single state.

After step 2, each of the superposition's component states has amplitude $1/\sqrt{N}$. As we will explain below, each iteration of the Grover iterate increases the amplitude of the state corresponding to the solution. So, by the end of step 3, the desired state has amplitude almost 1, and all other states have amplitude almost 0. The behavior of step 3 can be explained at several levels of mathematical detail. Below, we follow Section 4 of [Gro96], which is the most intuitive. For more precision, see [dW01, Section 1.7] and [BBHT98].

As already discussed, the Grover iterate consists of a query, which inverts the amplitude of the solution state, and the diffusion transform $(-I + [2/N])$. The diffusion transform achieves

[5] In other words, a total problem does not come with a promise that its input is either one way or another way.

[6] This subsection summarizes [Gro96], with some additions from [dW01].

what Grover calls "inversion about average." To see why, first note that applying the matrix $[1/N]$ to a vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ yields a vector whose components all equal the average A of the components of $\vec{v}$. So, applying the matrix $(-I + [2/N])$ to $\vec{v}$ yields the vector $\vec{w}$ whose ith component is $(-v_i + 2A) = (A + (A - v_i))$. That is, each component of $\vec{w}$ is as much below (or above) A as it was previously above (or below) it. So, $\vec{w}$ is the "inversion about average" of $\vec{v}$.

Inversion about average works considerable magic in step 3: After the first query, the desired state has amplitude $-1/\sqrt{N}$, and all other states have amplitudes $1/\sqrt{N}$. Clearly, the average amplitude A is slightly less than $1/\sqrt{N}$. So, after inverting about the average, the desired state has amplitude $O(2/\sqrt{N})$, and all other states have amplitudes just under $1/\sqrt{N}$. We make another query, which flips the amplitude of the desired state to $O(-2/\sqrt{N})$, and we perform another inversion about the average, which increases the amplitude of the desired state to $O(3/\sqrt{N})$ and further decreases all other amplitudes. The process continues, and after $O(\sqrt{N})$ queries and inversions about average, the amplitude of the desired state has grown to almost 1, and the other amplitudes have been whittled down to almost 0. So, when we measure the system in step 4, we observe the state corresponding to the solution with probability greater than 1/2. And we have achieved this with just $O(\sqrt{N})$ queries and just $O(\sqrt{N})$ operations.

Suppose that the input contains all 0s. We can learn that there are no solutions by running the algorithm and checking the value of the input bit indexed by the observed state. We see that we have not found a solution, and we can infer that there is no solution. (We can increase our confidence that there is no solution by running the algorithm several independent times.)

Now suppose that the input contains t bits that equal 1. It can be shown (again, see [dW01, Section 1.7] and [BBHT98]) that finding one of those solutions with high probability requires only $O(\sqrt{N/t})$ queries and operations, even if we do not know the value of t. And, to find all t solutions, we simply find one solution, set that solution bit to 0, find another solution, set that solution bit to 0, and so on. The query complexity and time complexity of this process are $\sum_{i=1}^{t} O\left(\sqrt{N/i}\right)$, which equals:

$$O\left(\sqrt{N} \cdot \sum_{i=1}^{t} \frac{1}{\sqrt{i}}\right) \leq O\left(\sqrt{N} \cdot \int_0^t \frac{1}{\sqrt{x}} dx\right) = O\left(\sqrt{N} \cdot \left[2\sqrt{x}\right]_0^t\right) = O\left(\sqrt{tN}\right).$$

2.3.2 Amplitude Amplification Algorithm

Grover's search algorithm was generalized into a quantum amplitude amplification algorithm by Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp [BHMT02]. Instead of finding a 1 in a bit string, amplitude amplification finds an x such that $\chi(x) = 1$, where χ is a Boolean function. As we will see, this allows amplitude amplification to fit into many different algorithmic contexts. We summarize [BHMT02] below.

Suppose we have a Boolean function $\chi : X \to \{0, 1\}$, where $x \in X$ is "good" if $\chi(x) = 1$ and "bad" otherwise. Suppose also that we have a quantum algorithm (i.e., a sequence of quantum gates) $\mathcal{A}$ such that $\mathcal{A}|0\rangle = \sum_{x \in X} \alpha_x |x\rangle$, a superposition of states $x \in X$ with amplitudes α_x. Let a denote the probability that we observe a good state if we measure $\mathcal{A}|0\rangle$. Let Q denote the iterate $-\mathcal{A}S_0\mathcal{A}^{-1}S_\chi$, where:

- $\mathcal{A}^{-1}$ is the inverse of $\mathcal{A}$;
- $S_0|\vec{0}\rangle = -|\vec{0}\rangle$, and S_0 does nothing to all other states;
- $S_\chi|x\rangle = (-1)^{\chi(x)}|x\rangle$.

The iterate Q is embedded in the amplitude amplification algorithm in the same way the Grover iterate is embedded in Grover's search algorithm:

Amplitude Amplification Algorithm

1. Begin in the n-qubit state $|\vec{0}\rangle$.
2. Apply $\mathcal{A}$ to $|\vec{0}\rangle$, resulting in the superposition $\sum_{x \in X} \alpha_x |x\rangle$. (If we measured the superposition now, the probability of observing a good state would be a.)
3. Apply Q to the superposition $O(1/\sqrt{a})$ times.
4. Measure the resulting superposition, collapsing it into a single state.

The analysis in [BHMT02] is similar to the analysis of Grover's algorithm and shows that we observe a good state in step 4 with probability close to 1. Moreover, it shows that if we know the value of a ahead of time, we observe a good state with certainty. Whether or not we know a, amplitude amplification performs just $O(1/\sqrt{a})$ queries and operations.

In algorithms, amplitude amplification can replace "for" loops, which step through every x in search of a good one. Indeed, many algorithms in the coming chapters use amplitude amplification to find good vertices and edges—that is, vertices and edges that appear in cycles.

It is easy to confirm that Grover's algorithm is a special case of amplitude amplification. Amplitude amplification becomes Grover's search when $\chi(i) = x_i$ and both $\mathcal{A}$ and $\mathcal{A}^{-1}$ are

the Hadamard transform H. And just as amplitude amplification is exact when a is known, Grover's algorithm can be modified into an exact algorithm when the number of solutions t is known.

2.3.3 Ambainis's Algorithm[7]

In 2003, Andris Ambainis developed an important quantum algorithm for the *element distinctness problem*, which is defined as follows: Given numbers $x_1, x_2, \ldots, x_N \in [M]$, where $N \leq M$, are there distinct $i, j \in [N]$ such that $x_i = x_j$? His algorithm generalizes for the *element k-distinctness problem*, which asks whether there are distinct $i_1, i_2, \ldots, i_k \in [N]$ such that $x_{i_1} = x_{i_2} = \ldots = x_{i_k}$. On the standard element distinctness problem, Ambainis's algorithm makes $O(N^{2/3})$ queries, which is a significant improvement over both classical algorithms (which require $O(N)$ queries) and previous quantum algorithms (the best of which makes $O(N^{3/4})$ queries). On the k-distinctness problem, Ambainis's algorithm makes $O(N^{k/(k+1)})$ queries.

Ambainis's algorithm involves searching, but it uses neither Grover's algorithm nor amplitude amplification. Instead, it examines a sequence of subsets $S \subseteq [N]$ such that $|S| = N^{2/3}$ or $|S| = N^{2/3} + 1$. More specifically, Ambainis's algorithm for standard element distinctness reduces to finding a marked vertex in the following graph: Let $r = N^{2/3}$, and let G be a graph with $\binom{N}{r} + \binom{N}{r+1}$ vertices. The vertices v_S correspond to subsets S of $[N]$ of size r and $r+1$. Two vertices v_S and v_T are connected by an edge if $T = S \cup \{i\}$ for some $i \in [N]$. That is, v_S and v_T are connected if $|S| = r$, $|T| = r + 1$, and $T \supset S$. A vertex v_S is marked if and only if S contains distinct i and j such that $x_i = x_j$. Finding a marked vertex v_S reveals that $x_i = x_j$ for some $i, j \in S$ and thus that $x_1, x_2, \ldots, x_N$ are not all distinct.

Ambainis's algorithm searches this graph G using quantum walks, which are explained at length in numerous articles (for example, [Kem03] and [AAKV01]). Broadly, a quantum walk determines whether a set satisfies a certain property by stepping through (in superposition) the set's subsets of certain sizes. Each subset contains just one thing that its predecessor lacked (or is missing just one thing that its predecessor had). Therefore, any visited subset is very similar to its predecessor, and computation on that subset is very similar to computation on its predecessor.

More formally, the state of a quantum walk is stored in two registers, the *node register*

[7]This subsection summarizes [Ambb], with some additions from [MSSb].

and the *coin register*. The node register holds S, a subset of $[N]$ of size either r or $r+1$. The coin register holds an element $y \in [N]$. There is a condition on y, though: If $|S| = r$, then $y \notin S$, but if $|S| = r+1$, then $y \in S$.

A quantum walk steps through a sequence of these subsets S. Each step of the walk consists of the following operations:

A Single Step of Quantum Walk

1. Diffuse the coin register $|y\rangle$ over $[N] - S$.[8]
2. Add y to S (increasing the size of S from r to $r+1$).
3. Query for the value of x_y.
4. Diffuse the coin register $|y\rangle$ over S.
5. Undo the query for x_y.
6. Remove y from S (decreasing the size of S from $r+1$ to r).

Suppose we have a promise that either $x_1, x_2, \ldots, x_N$ are all distinct or that there are exactly two elements i, j such that $i \neq j$ and $x_i = x_j$. This promise problem is solved by an algorithm that embeds steps of the quantum walk as follows:

Ambainis's Element Distinctness Algorithm

1. Put node register $|S\rangle$ in uniform superposition of all subsets of $[N]$ of size r, and put coin register $|y\rangle$ in uniform superposition of all elements not included in each S.
2. Query x_i for all $i \in S$.
3. Repeat $O(N/r)$ times:
4. For subsets S that contain distinct elements i and j such that $x_i = x_j$, flip the phase from $|S\rangle|y\rangle$ to $-|S\rangle|y\rangle$.
5. Perform $O(\sqrt{r})$ steps of the quantum walk.
6. Measure the resulting superposition, collapsing it into a single state.

Step 1 of this algorithm makes no queries, but step 2 makes $r = N^{2/3}$ queries. Step 5 performs $O(\sqrt{r})$ steps of the quantum walk and therefore $O(\sqrt{r}) = O(\sqrt{N^{2/3}}) = O(N^{1/3})$ queries. The loop in steps 3–5 makes $O(N^{1/3})$ queries per iteration, and therefore $O(N/r)O(N^{1/3}) =$

[8] Diffusion "over" something is closely related to diffusion as it was introduced in Section 2.3.1. The *diffusion transform over* T (where T is a finite set) acts on T's basis elements $|x\rangle$ as follows: $|x\rangle \mapsto -|x\rangle + \frac{2}{|T|}\sum_{y \in T} |y\rangle$.

$O(N/N^{2/3})O(N^{1/3}) = O(N^{2/3})$ queries total. Step 6 makes no queries, so the overall query complexity of the algorithm is $O(N^{2/3})$.

To see how this algorithm generalizes for total versions of the element distinctness problem and the k-element distinctness problem, refer to [Ambb]. Also, see [Ambb] for analyses of the correctness of all of Ambainis's element distinctness algorithms. These analyses are well beyond the scope of this paper; for our purposes, it suffices to note that Ambainis's algorithm bears some striking resemblances to Grover's algorithm.

2.3.4 Embedding Quantum Algorithms into Larger Algorithms

As mentioned before, Grover's search algorithm, amplitude amplification, and Ambainis's algorithm make up our "bag of tricks." In the coming chapters, we will combine them and embed them with each other and with classical operations to create quantum cycle algorithms. Sometimes, we can speed up an algorithm simply by replacing some classical steps with one of our tricks. Other times, a simple replacement has no effect on the algorithm's efficiency because the remaining classical operations dominate the algorithm's complexity. In these cases, the algorithm needs to be rebalanced or otherwise reconceived before it can benefit from quantum interventions.

2.4 A Hypothetical Situation

Now is a good time to ask "What if?" What if we had a decision-only algorithm for the element distinctness problem? How could we transform it into an example-finding algorithm, and how much less efficient would the new algorithm be? A good strategy, we find, would be to embed the algorithm in a recursion.

Let us formalize. Suppose we are working with a quantum element distinctness decision algorithm whose success probability is at least 2/3. The quantum lower bound for the element distinctness problem is $\Omega(N^{2/3})$ queries [Amba], so suppose that our algorithm's query complexity is $O(N^{2/3})$.[9]

Suppose also that this algorithm takes as input a list L with length $N = 2^n$ and returns

[9]We could just as easily be working with a classical element distinctness decision algorithm. If we were, its time complexity would be bounded below by $\Omega(N \log N)$ [BDH$^+$01]. Its query complexity would be bounded above and below by $\Theta(N)$. (Every element may need to be queried, and no element needs to be queried more than once.)

either "all distinct" or "not all distinct." (If L's length is not a power of 2, pad L so that it is. The padding must not affect the outcome of the algorithm, so the padding elements must be distinct from the pre-existing list elements and distinct from each other.) Then we can do the following:

Decision-to-Example-Finding Element Distinctness Algorithm

1. Run element distinctness decision algorithm on L.
2. If algorithm returns "all distinct," return "all distinct." Otherwise, return results of Example-Finding Procedure.

Example-Finding Procedure

1. If $|L| = 2$, then L's two elements are indistinct. Return their indices.
2. Randomly reorder the elements in L, keeping a record of each element's original index.
3. Split L into two segments S_1 and S_2, each with $N/2$ elements.
4. Run element distinctness decision algorithm on S_1. If algorithm returns "all distinct," return with failure. Otherwise, recur on S_1. If recursion returns indices, return original indices of the elements identified by the returned indices. Otherwise, return with failure.

First, let us analyze the success probability of the Example-Finding Procedure. Each level of the procedure's recursion involves randomly rearranging the elements of the inputted list L, halving L into segments S_1 and S_2, and running the decision algorithm for element distinctness on S_1. After each random rearrangement, the probability that two indistinct elements are in S_1 is at least $1/4$. Since the success probability of the decision algorithm is at least $2/3$, the success probability of each recursive level is at least $1/6$ (and the failure probability of each level is at most $5/6$).

For overall success in the Example-Finding Procedure, we must succeed on each of the $\log N$ recursive levels. Therefore, the overall success probability of the procedure is at least $(1/6)^{\log N}$. This probability is not good enough, but we can make it arbitrarily close to 1 by performing a sufficient number of independent repetitions of each recursive level, according to the argument below.

Suppose that each level of the recursion is repeated up to $k+j$ times, where k is a constant and j is the level of the recursion (and therefore between 0 and $\log N - 1$). In other words, suppose that recursive level j does not return with failure until the decision algorithm for

element distinctness has failed on $k+j$ different segments S_1. The failure probability of each repetition is at most 5/6, so the failure probability of each level is at most $(5/6)^{k+j}$.

Now we need to determine the overall failure probability of the Example-Finding Procedure, where overall failure is caused by failure at any of the $\log N$ recursive levels. Because the probability of a union is less than or equal to the sum of the probabilities of each component, the overall failure probability is less than or equal to:

$$\sum_{j=0}^{\log N-1} \left(\frac{5}{6}\right)^{k+j} = \sum_{j=0}^{\log N-1} \left(\frac{5}{6}\right)^{k} \left(\frac{5}{6}\right)^{j} < \sum_{j=0}^{\infty} \left(\frac{5}{6}\right)^{k} \left(\frac{5}{6}\right)^{j},$$

which is an infinite geometric series and therefore converges to:

$$\frac{\left(\frac{5}{6}\right)^{k}}{1-\frac{5}{6}} = 6\left(\frac{5}{6}\right)^{k}.$$

However small we want the overall failure probability to be, there is a k that makes the failure probability that small. For example, suppose we want the overall failure probability of the Example-Finding Procedure to be less than 1/10. If $k = 25$, then the failure probability of the Example-Finding Procedure is at most $6(5/6)^{25} = 5^{25}/6^{24} < 0.063$, which is less than 1/10.

Of course, the Decision-to-Example-Finding Algorithm involves not just the Example-Finding Procedure but also one run of the element distinctness decision algorithm. The failure probability of the decision algorithm is at most 1/3, so the overall failure probability of the Decision-to-Example-Finding Algorithm is less than $0.33\overline{3} + 0.063 = 0.396\overline{3}$, which is acceptably low.

Now let us analyze the complexity of the Example-Finding Procedure. Again, the decision algorithm runs up to $k+j$ times on each of up to $\log N$ levels, where each level's input is half as long as the input on the previous level. Therefore, the query complexity of the Example-Finding Algorithm is:

$$O\left(\sum_{j=0}^{\log N-1} \left(\frac{N}{2^j}\right)^{\frac{2}{3}} \cdot (k+j)\right) = O\left(N^{2/3} \cdot \sum_{j=0}^{\log N-1} \frac{k+j}{(2^{2/3})^j}\right) < O\left(N^{2/3} \cdot \sum_{j=0}^{\infty} \frac{k+j}{(2^{2/3})^j}\right).$$

Because $k+j$ grows linearly while $(2^{2/3})^j$ grows exponentially, the sum in the expression above converges to a constant. Therefore, the query complexity of the Example-Finding Procedure, and in fact the Decision-to-Example-Finding Algorithm, is $O(N^{2/3})$. So, despite the fact

that it contains a procedure that runs the decision algorithm multiple times, the Decision-to-Example-Finding Algorithm has the same query complexity as the decision algorithm.

2.5 Moving On

We now know enough about classical and quantum computing to understand the appeal of embedding quantum search (and other quantum algorithms) in graph algorithms—and to understand the difficulty of improving the algorithms' complexities. Thus equipped, we devote the rest of this paper to examining classical and quantum algorithms for cycles.

Chapter 3

Triangle Algorithms

In undirected graphs, the simplest cycles are 3-cycles, often called *triangles*. Determining whether a graph $G = (V, E)$ contains a triangle is a straightforward task, and most triangle algorithms are short and simple. We discuss three classical algorithms and six quantum algorithms—presented in order of increasing sophistication.[1] All but Classical Triangle Algorithm #3 are example-finding algorithms.[2]

Remember that we care about the query complexities of quantum algorithms. So, in order to make meaningful comparisons between quantum and classical algorithms, we calculate the query complexities of classical algorithms as well. But, in order to understand the classical algorithms as fully as possible, we also discuss their time complexities. Both kinds of complexity depend on the representation of the input graph, so we always note whether an algorithm uses an adjacency matrix, an adjacency list, or both.

3.1 Classical Triangle Algorithms

3.1.1 Classical Triangle Algorithm #1

The first classical triangle algorithm uses an adjacency matrix M and is an exhaustive search:

[1] For the sake of less experienced readers, we move slowly through this chapter's algorithms and analyses. We quicken our pace in Chapters 4 and 5.

[2] Our selection of triangle algorithms is by no means complete. Nor are our selections of algorithms in Chapters 4 and 5. Many more given-length cycle algorithms (some of which involve graph properties not explained in this paper) are discussed in [AYZ97].

Classical Triangle Algorithm #1

1. For each triple of distinct vertices $u, v, w \in V$,
2. If $M(u, v) = M(v, w) = M(w, u) = 1$, then u, v, w is a triangle.

There are $\binom{n}{3} = n(n-1)(n-2)/6 \in O(n^3)$ potential triangles in G, and checking whether a potential triangle is an actual triangle requires three matrix queries. Therefore, the algorithm's time complexity is $O(n^3)$. However, matrix queries do not need to be repeated, so the query complexity is bounded by the size of the matrix: $O(n^2)$.

3.1.2 Classical Triangle Algorithm #2

An exhaustive search through an adjacency list A is no more efficient:

Classical Triangle Algorithm #2

1. For each vertex $u \in V$,
2. Make a temporary array T of length n and initialize its elements to 0.
3. For each neighbor $v \in A[u]$,
4. Set $T[v] \leftarrow 1$.
5. For each vertex $v \in A[u]$,
6. For each vertex $w \in A[v]$,
7. If $T[w] = 1$, then u, v, w is a triangle.

The "for" loops in lines 1, 5, and 6 dominate the algorithm. Each loop steps through as many as n vertices, and step 7 takes constant time, so the time complexity is $O(n^3)$. The algorithm might query every element of A, so the query complexity is $O(n + m) = O(n^2)$.

3.1.3 Classical Triangle Algorithm #3

The third classical algorithm [vL90, page 563] is a decision algorithm and consists entirely of matrix manipulations:

Classical Triangle Algorithm #3

1. Square adjacency matrix M using Boolean matrix multiplication.
2. Compute $M^2 \wedge M$.
3. If $M^2 \wedge M$ contains a 1, then G contains a triangle.

Step 1 computes the Boolean matrix M^2, which contains a 1 if and only if G has a path of length 2. Step 2 computes $M^2 \wedge M$, which contains a 1 if and only if the first and third vertices of a path of length 2 are adjacent. So, the final matrix contains a 1 if and only if G has a triangle.

It is harder to multiply matrices than to AND them, so this algorithm's time complexity depends on the difficulty of Boolean matrix multiplication. A naive Boolean matrix multiplication algorithm performs $O(n^3)$ operations, but there are asymptotic improvements. One improvement is the Boolean version of Strassen's algorithm, which performs $O(n^{\lg 7}) \approx O(n^{2.81})$ operations [CLR90, page 748]. Another is by Coppersmith and Winograd and performs only $O(n^{2.376})$ operations [CW90].

So, the time complexity is $O(n^\alpha)$, where α is the exponent of the Boolean matrix multiplication algorithm we employ. The query complexity is obvious: We must know every element of the adjacency matrix, so we must make $O(n^2)$ queries.[3]

Clearly, this algorithm does not involve a search—or any task that can reasonably be transformed into a search. Therefore, only the first and second classical algorithms can benefit from Grover's algorithm, and we now examine how.

3.2 Quantum Triangle Algorithms

3.2.1 Quantum Triangle Algorithm #1

The first quantum algorithm [BDH+01] is exactly like the first classical algorithm, except that Grover's search replaces the exhaustive search through all potential triangles:

Quantum Triangle Algorithm #1

1. Perform Grover's search to find a triangle among the $\binom{n}{3}$ triples of vertices.

If a triangle exists among the $\binom{n}{3} \in O(n^3)$ triples, Grover's algorithm finds it with probability at least 1/2 in only $O(\sqrt{n^3}) = O(n^{3/2})$ queries. This is a notable improvement over the classical version's $O(n^2)$ queries.

[3]Because Classical Triangle Algorithm #3 finds an edge of any triangle that it detects, it can easily be extended into an example-finding algorithm. All that is needed is to look through the original matrix M for a vertex that is adjacent to both ends of the known edge. Needless to say, this extra step does not increase the algorithm's time or query complexity.

3.2.2 Quantum Triangle Algorithm #2

The second quantum algorithm [BDH+01] introduces amplitude amplification. It too uses only an adjacency matrix:

Quantum Triangle Algorithm #2

1. Perform Grover's search to find an actual edge $(u, v) \in E$ among the $\binom{n}{2}$ potential edges.
2. Perform Grover's search to find a vertex $w \in V$ such that u, v, w is a triangle.
3. Perform amplitude amplification on steps 1 and 2.

There are $\binom{n}{2} = n(n-1)/2 \in O(n^2)$ possible edges and m actual edges, so step 1 requires $O(\sqrt{n^2/m})$ queries. Step 2 requires $O(\sqrt{n})$ queries. Therefore, the complexity of steps 1–2 is $O(\sqrt{n^2/m} + \sqrt{n})$.

However, steps 1–2 are not sufficient. If step 1 finds a triangle edge, the probability that step 2 finds the triangle's third vertex is at least 1/2. However, when G contains only one triangle, the probability that step 1 finds a triangle edge is only $O(1/m)$, which means that steps 1–2 find a triangle with probability only $O(1/m)$.[4] This probability can be boosted with $O(\sqrt{m})$ iterations of amplitude amplification. Thus, the overall query complexity of the algorithm is $O\left(\sqrt{m}(\sqrt{n^2/m} + \sqrt{n})\right) = O\left(n + \sqrt{nm}\right)$. This complexity is better than $O(n^{3/2})$ if G is sparse.

3.2.3 Quantum Triangle Algorithm #3

The third quantum algorithm [BdW] is an adaptation of the second classical algorithm and uses an adjacency list only:

Quantum Triangle Algorithm #3

1. Choose a random vertex $u \in V$.
2. Query all elements in $A[u]$ and make a temporary list T of these neighbors.
3. Choose a random vertex $v \in T$.
4. Perform Grover's search to find a vertex $w \in A[v]$ that is also in T. If such a w is found, then u, v, w is a triangle.
5. Perform amplitude amplification on steps 3 and 4 (to find a desired v).
6. Perform amplitude amplification on steps 1–5 (to find a desired u).

[4] The probability is greater than $O(1/m)$ when G contains more than one triangle.

This algorithm is unlike the others in this paper because it contains two layers of amplitude amplification. Therefore, we compute its query complexity from the outside in. Step 1 chooses one of n vertices, so steps 1–5 must be embedded in $O(\sqrt{n})$ iterations of amplitude amplification (step 6). In other words, the algorithm's overall query complexity is $O(\sqrt{n})$ times the query complexity of steps 2–5.

Step 2 queries up to $n-1$ elements of A to create T (which, after its creation, is fully known and costs nothing to query). Step 3 chooses one of up to $n-1$ vertices, so steps 3 and 4 must also be embedded in $O(\sqrt{n})$ iterations of amplitude amplification (step 5). The search in step 4 makes $O(\sqrt{n})$ queries, so steps 3–5 make $O(n)$ queries. Therefore, steps 2–5 make $O(n)$ queries, and steps 1–6 make $O(n^{3/2})$ queries.

3.2.4 Quantum Triangle Algorithm #4

In 2003, Mario Szegedy announced a quantum triangle algorithm with a query complexity of only $O(n^{10/7} \log^2 n)$ [Sze].[5] The algorithm's only quantum trick is Grover's search algorithm, but its combinatorial tricks are many, and it is very difficult to understand. Furthermore, its was almost immediately overshadowed by a $\widetilde{O}(n^{1.3})$ algorithm by Magniez, Santha, and Szegedy [MSSb]. Therefore, we skip the $\widetilde{O}(n^{10/7})$ algorithm and focus instead on the $\widetilde{O}(n^{1.3})$ one.

3.2.5 Quantum Triangle Algorithm #5

Unlike the Szegedy algorithm, the Magniez, Santha, and Szegedy algorithm uses quantum sophistication, not combinatorial sophistication [MSSb]. But, rather than lay out an actual triangle algorithm, they show that the triangle problem reduces to the "graph collision" problem, which reduces to the "collision" problem, which reduces to the "unique collision" problem, which can be solved by a generalization of Ambainis's Element Distinctness Algorithm.

First, let's summarize these intermediate problems:

- The *collision problem* supplies a function f that defines a relation $C \subseteq [N]^2$, and asks whether C is non-empty. If so, a pair $(a, b) \in C$ should be returned; otherwise, an algorithm should return "reject."

[5] Here and elsewhere, we can use the $\widetilde{O}$ notation, which allows us to ignore complexities' logarithmic factors. In this case, we can write $O(n^{10/7} \log^2 n)$ as $\widetilde{O}(n^{10/7})$.

- The *unique collision problem* is the same as the collision problem but promises that either $|C| = 1$ or $|C| = 0$.

- The *graph collision problem* is the same as the collision problem except that f is a boolean function on V, and it defines the relation $C \subseteq V^2$ such that $(u, v) \in C$ if and only if $f(u) = f(v) = 1$ and $(u, v) \in E$.

As said above, the unique collision problem can be solved by a generalization of Ambainis's Algorithm. This "Generic Algorithm" has three registers: $|S\rangle|D(S)\rangle|y\rangle$, where $|S\rangle$ is the node register (familiar from Ambainis's algorithm), $|y\rangle$ is the coin register (also familiar from Ambainis's algorithm), and $|D(S)\rangle$ is the data register (new to this algorithm). The *data register* holds data $D(S)$ about the set S. (This data comes from a database D, which associates data with every subset of $[N]$.)

We want to use $D(S)$ to determine whether $(S \times S) \cap C \neq \emptyset$. To do this, we use a quantum checking procedure Φ such that $\Phi(D(S))$ rejects if $(S \times S) \cap C = \emptyset$ and otherwise outputs an element of $(S \times S) \cap C$. Using the database D incurs three kinds of costs:

- Setup cost — $s(r)$ — the query complexity of setting up $D(S)$, where $|S| = r$. In the collision problems described above, $s(r) = r$.

- Update cost — $u(r)$ — the query complexity of updating $D(S)$ to $D(S')$, where S' is the result of adding an element to S (if S is of size r) or deleting an element from S (if S if of size $r + 1$). In the collision problems described above, $u(r) = 1$.

- Checking cost — $c(r)$ — the query complexity of finding $\Phi(D(S))$, where $|S| = r$. In the collision problems described above, $c(r) = 0$.

We will consider these costs when we evaluate the complexity of the Generic Algorithm. First though, we present the Generic Algorithm, which solves the unique collision problem:

Generic Algorithm

1. Put the node register $|S\rangle$ in uniform superposition of all subsets of $[N]$ of size r.
2. Set up D on S in the data register.
3. Put the coin register $|y\rangle$ in uniform superposition of all elements of $[N] - S$.
4. Do $\Theta(N/r)$ times:
5. Check $\Phi(D(S))$. If it accepts, then flip the phase from $|S\rangle|D(S)\rangle|y\rangle$ to $-|S\rangle|D(S)\rangle|y\rangle$; otherwise do nothing.
6. Perform $\Theta(\sqrt{r})$ steps of the quantum walk algorithm, updating the data register with each alteration of S.
7. Measure the system and check $\Phi(D(S))$. It it rejects, then reject; otherwise output the collision given by $\Phi(D(S))$.

Steps 1 and 3 require no queries, but step 2 requires $s(r)$ queries. Step 4 requires $\Theta(N/r)$ repetitions of step 5 (which requires $c(r)$ queries) and step 6 (which requires $u(r)$ queries $\Theta(\sqrt{r})$ times). Like step 5, step 7 requires $c(r)$ queries. These steps total to $O(s(r)+\frac{N}{r}(c(r)+(\sqrt{r}\times u(r)))+c(r))$ queries. We know that for the collision problem, $s(r) = r$, $u(r) = 1$, and $c(r) = 0$. Therefore, the query complexity of the Generic Algorithm is: $O(r+\frac{N}{r}(0+1\sqrt{r})+0)$. When $r = N^{2/3}$, this complexity equals $O(N^{2/3} + N^{1/3}N^{1/3}) = O(N^{2/3})$.

The Generic Algorithm can be transformed from a unique collision algorithm to a collision algorithm in the same way that Ambainis's Element Distinctness Algorithm (as presented in Section 2.3.2) can be transformed into a total Element Distinctness Algorithm (again, see [Ambb]). This requires a logarithmic number of iterations of the Generic Algorithm, so the query complexity of the collision algorithm is $\widetilde{O}(N^{2/3})$.

The graph collision problem is a specialized version of the collision problem, and it can be solved with a specialized version of the collision algorithm in which $S = V$. Define on every subset $U \subseteq V$, $D(U) = \{(v, f(v)) : v \in U\}$, and let $\Phi(D(U)) = 1$ if there are two vertices $u, u' \in V$ that satisfy the graph collision property. As with the unique collision and collision problems, $s(r) = r$, $u(r) = 1$, and $c(r) = 0$. So, as with the collision problem, when $r = N^{2/3}$, graph collision can be solved with $\widetilde{O}(N^{2/3})$ queries.

Finally, the triangle problem is an extension of the graph collision problem. Again let $S = V$, but this time, for every subset $U \subseteq V$, we define $D(U) = G|_U$, where $G|_U$ is the subgraph of G generated by U. Furthermore, we define Φ such that $\Phi(G|_U) = 1$ if some triangle in G has an edge in $G|_U$.

With these definitions, the setup cost $s(r)$ is the query complexity of setting up $G|_U$, where $G|_U$ has $O(r^2)$ edges. Therefore, $s(r) = O(r^2)$. Updating subgraph $G|_U$ requires visiting each vertex, so the update cost $u(r)$ is r. Checking subgraph $G|_U$ involves looking for a triangle, which can be done as follows:

Let U be a set of r vertices, let v be a vertex in V, and let $f(u) = 1$ if (u, v) is an edge in G. Use the graph collision algorithm to look for edges $(u, u') \in G|_U$ such that both $f(u) = 1$ and $f(u') = 1$; since $G|_U$ has r vertices, performing this graph collision algorithm requires $\widetilde{O}(r^{2/3})$ queries. However, in order to find a "good" v, one that forms a triangle with an edge in $G|_U$, the graph collision algorithm must be embedded in the amplitude amplification algorithm. Because there are N vertices in V, there need to be $O(\sqrt{N})$ iterations of amplitude amplification. So, the amplified graph collision algorithm requires $\widetilde{O}(\sqrt{N} \times r^{2/3})$ queries. In other words, the cost $c(r)$ of checking a subgraph $G|_U$ for a triangle is $\widetilde{O}(\sqrt{N} \times r^{2/3})$ queries.

Recall that the triangle problem is ultimately a specialized version of the collision algorithm, which requires a logarithmic number of iterations of the Generic Algorithm. Recall also that each iteration of the Generic Algorithm requires $O\left(s(r) + \frac{N}{r}(c(r) + (\sqrt{r} \times u(r))) + c(r)\right)$ queries. Because $c(r)$ is always less than $\frac{N}{r}(c(r))$, the second $c(r)$ can be dropped, for a Generic Algorithm query complexity of $O\left(s(r) + \frac{N}{r}(c(r) + (\sqrt{r} \times u(r)))\right)$ and a quantum triangle algorithm query complexity of:

$$\widetilde{O}\left(s(r) + \frac{N}{r}\left(c(r) + (\sqrt{r} \times u(r))\right)\right) = \widetilde{O}\left(r^2 + \frac{N}{r}\left((\sqrt{N} \times r^{2/3}) + (\sqrt{r} \times r)\right)\right).$$

This time, instead of letting r equal $N^{2/3}$, let it equal $N^{3/5}$. Then the above query complexity equals:

$$\widetilde{O}\left((N^{3/5})^2 + \frac{N}{N^{3/5}}\left((\sqrt{N} \times (N^{3/5})^{2/3}) + (\sqrt{N^{3/5}} \times N^{3/5})\right)\right) =$$

$$\widetilde{O}\left(N^{6/5} + N^{2/5}\left((N^{1/2} \times N^{2/5}) + (N^{3/10} \times N^{3/5})\right)\right) =$$

$$\widetilde{O}\left(N^{6/5} + N^{2/5}(N^{9/10} + N^{9/10})\right) = \widetilde{O}\left(N^{6/5} + N^{13/10}\right) = \widetilde{O}\left(N^{13/10}\right).$$

Shortly after releasing this algorithm, Magniez, Santha, and Szegedy showed that preprocessing can improve this triangle algorithm's complexity from $\widetilde{O}(n^{1.3})$ queries to $O(n^{1.3})$ queries. Consult [MSSa] for details.[6]

[6] In the summarizing table in Chapter 6, we refer to this improved $O(n^{1.3})$ algorithm as Quantum Triangle Algorithm #5I.

3.3 Another Hypothetical Situation

It is again a good time to ask "What if?" What if we had a decision-only algorithm for the triangle problem? Could we transform it into an example-finding algorithm in the same way that we can transform a decision-only algorithm for the element distinctness problem? Let us see:

Suppose we have a quantum triangle decision algorithm whose query complexity is $O(n^\beta)$ and whose success probability is at least $2/3$.[7] Suppose also that its input is an adjacency matrix M of graph G. There are two potential obstacles to embedding this algorithm in a recursion as we did the element distinctness algorithm in Section 2.4. First, we need a way of keeping track of the "active" vertices at each level of recursion. This is not a problem: We can modify the triangle algorithm so that it also takes in a simple, one-dimensional list of vertices. Like the element distinctness algorithm's list of elements, this list is easy to reorder and divide.

The second potential obstacle is that the graph shrinks each time the vertices are split in half, and we need a way of passing the correct adjacency matrix to each recursive call. This is not a problem either: We always pass the entire, original adjacency matrix. The decision algorithm knows which vertices are active and queries only matrix elements corresponding to active vertices.

So, we do not have any significant obstacles preventing us from embedding the decision algorithm in a recursion. Suppose that the decision algorithm returns either "contains no triangle" or "contains triangle." Then we can embed the algorithm as follows:

Decision-to-Example-Finding Triangle Algorithm

1. Make list L of vertices in graph G.
2. Run triangle decision algorithm on adjacency matrix M and vertex list L.
3. If algorithm returns "contains no triangle," return "contains no triangle." Otherwise, return results of Example-Finding Procedure.

[7] Again, we could be working with a classical algorithm with a different success probability. If we were, the analysis would differ slightly.

Example-Finding Procedure

1. If $|L| = 4$, then L's four vertices must contain a triangle. Find the triangle and return its vertices. If $|L| > 4$, go to step 2.
2. Randomly reorder the vertices in L.
3. Split L into two segments S_1 and S_2, each with $n/2$ vertices.
4. Run triangle decision algorithm on M and S_1. If algorithm returns "contains no triangle," return with failure. Otherwise, recur on M and S_1. If recursion returns vertices, return the vertices; if not, return with failure.

Again, the success probability of this Example-Finding Procedure is not good enough. Fortunately, the argument from Section 2.4 can be reapplied here: Repeating each recursive level $k + j$ times, where k is a constant and j is the level of the recursion, boosts the success probability sufficiently high and allows the Decision-to-Example-Finding Triangle Algorithm to maintain its query complexity of $O(n^\beta)$.

So, yes, the method from Section 2.4 can be applied to a triangle decision algorithm that uses an adjacency matrix. Now suppose we have a triangle decision algorithm whose input includes an adjacency list. Can we do what we did with the adjacency matrix algorithm and pass the entire adjacency list to each recursive call? No, we cannot, as doing so would endanger the overall algorithm's correctness. For example, the decision algorithm might run an element distinctness algorithm on the adjacency list. If the list's active vertices were all distinct but its inactive vertices were not, the element distinctness algorithm would return an incorrect answer.

Thus, we would need to prune the adjacency list before recurring. Doing this would require up to $O(n^2)$ queries before the first recursive call, up to $O((n/2)^2)$ queries before the second recursive call, up to $O((n/4)^2)$ queries before the third recursive call, and so on—for a total of $O(n^2)$ queries. So, the query complexity of the Example-Finding Procedure would be $O(n^2)$, even if the query complexity of the decision algorithm were considerably lower. In conclusion, we could embed the decision algorithm in a recursion, but the query complexity of the "upkeep" of the adjacency list would dominate the query complexity of the decision algorithm.

3.4 Triangle Lower Bounds

Different triangle algorithms have different upper bounds, but the triangle problem itself has algorithm-independent lower bounds. We can determine these lower bounds by reducing the OR problem to the triangle problem [BDH$^+$01]:

Let $X \in \{0,1\}^{\binom{n}{2}}$ be an input to the OR problem. Think of X as a graph G on n vertices, and think of the bits in X as indicators about which of the graph's $\binom{n}{2}$ possible edges exist. Now, add a vertex to G, connect it to every other vertex in G, and call this supergraph G'. Clearly, G' has a triangle if and only if G has at least one edge—that is, if and only if $\mathrm{OR}(X) = 1$.

So, the OR problem reduces to the triangle decision problem. Thus, deciding whether an $(n+1)$-vertex graph contains a triangle requires at least as many queries as computing the OR of $\binom{n}{2}$ bits. Classically, the OR problem requires as many queries as its input has bits, so the classical lower bound of the triangle problem is $\binom{n}{2} \in \Omega(n^2)$ queries. Quantumly, the N-bit OR problem requires $O(n)$ queries, so the quantum lower bound of the triangle problem is $\Omega(n)$ queries.

Chapter 4

Quadrilateral Algorithms

We now turn to the problem of determining whether an undirected graph contains a 4-cycle, often called a *quadrilateral*. Not surprisingly, many quadrilateral algorithms are reworkings of triangle algorithms. We look at some of these derivative algorithms, but we also study some that are quite unlike any of the triangle algorithms. All of the algorithms in this chapter are example-finding algorithms.

4.1 Classical Quadrilateral Algorithms

4.1.1 The Obvious Adaptations

Classical Triangle Algorithm #1 can be adapted for quadrilaterals very simply: Change the search through $\binom{n}{3}$ potential triangles into a search through $\binom{n}{4}$ potential quadrilaterals. This new search would take $O(n^4)$ time and $O(n^2)$ queries.[1]

Classical Triangle Algorithm #2 can be revised for quadrilaterals in several ways. The original algorithm looks through combinations of an edge and a vertex, and we can modify it to look through combinations of an edge and two vertices, combinations of two edges, or combinations of a path of length two and a vertex. None of these revisions are interesting or efficient enough to warrant careful study here.

It is tempting to modify Classical Triangle Algorithm #3 for quadrilaterals by computing $M^3 \wedge M$. And indeed, $M^3 \wedge M$ reveals whether G has a path $\langle v_0, v_1, v_2, v_3, v_0 \rangle$. However, that is not the same as revealing whether G has a quadrilateral, as there is no guarantee

[1]There is a quantum version of this algorithm too. However, its query complexity is $O(n^{4/2}) = O(n^2)$, which is no better than the classical algorithm's query complexity. Therefore, we do not present it in Section 4.2.

that v_1 is distinct from v_3. That said, Classical Triangle Algorithm #3 can be adapted for quadrilaterals and k-cycles. The adaptation is explained in Section 5.1.4.

4.1.2 Classical Quadrilateral Algorithm #1

Classical Quadrilateral Algorithm #1 [RL85] is faster than the classical triangle algorithms. It takes in an adjacency list A and cleverly creates a matrix:[2]

Classical Quadrilateral Algorithm #1

1. Create an $n \times n$ matrix C and initialize all entries to 0.
2. For each vertex $u \in V$,
3. For each pair of vertices $v, w \in A[u]$ such that $v < w$,
4. If $C(v, w) = 0$, then $C(v, w) \leftarrow u$. Else, $u, v, C(v, w), w$ is a quadrilateral.

Note that matrix C is not an adjacency matrix. Rather, it is a matrix of *corners*, or middle vertices in paths of length 2. The algorithm's mission is to find a pair of vertices between which are two different corners.

Each iteration of step 4 (except possibly the last one) finds one corner and changes an entry $C(v, w)$ from 0 to the corner's vertex value. If there is a quadrilateral, the algorithm ends when step 4 finds an entry of C that already contains a vertex value. If there is not a quadrilateral, the algorithm ends when all vertices $u \in V$ and all pairs $v, w \in A[u]$ such that $v < w$ have been exhausted. Because $v < w$, all changes occur in the $n(n-1)/2$ elements above C's main diagonal. Therefore, there can be at most $(n(n-1)/2)+1$ iterations of step 4, and the algorithm runs in $O(n^2)$ time. And, because A has $O(n+m)$ elements, the algorithm makes $O(n+m)$ queries.

4.2 Quantum Quadrilateral Algorithms

4.2.1 Quantum Quadrilateral Algorithm #1

Our first quantum quadrilateral algorithm [BdW] is a straightforward adjacency matrix algorithm:

[2]Classical Quadrilateral Algorithm #1 could take in an adjacency matrix M instead. If so, it would need a preliminary step that transforms M into an adjacency list. (The algorithm could be adapted for an adjacency matrix in other ways, but the other adaptations are no more efficient.) This extra step would take $O(n^2)$ queries, but the algorithm would still run in $O(n^2)$ time. In the summarizing table in Chapter 6, we refer to this variation as Classical Quadrilateral Algorithm #1M.

Quantum Quadrilateral Algorithm #1

1. Choose two random vertices $u, v \in V$.
2. Perform Grover's search to find two vertices $u', v' \in V$ such that u, u', v, v' is a quadrilateral.
3. Perform amplitude amplification on steps 1–2.

Step 1 selects random vertices u and v, and step 2 searches for vertices u' and v' that are adjacent to both u and v. This search requires $O(\sqrt{n})$ queries and succeeds with probability at least $1/2$ if u and v are in a four-cycle together.

However, the probability that u and v are in the same four-cycle is just $O(1/n^2)$, so the success probability of steps 1–2 is also just $O(1/n^2)$. We boost the algorithm's success probability by embedding steps 1–2 in $O(n)$ iterations of amplitude amplification. This raises the overall success probability to at least $1/2$ and raises the overall query complexity to $O(n^{3/2})$.

4.2.2 Quantum Quadrilateral Algorithm #2

Our second quantum quadrilateral algorithm [BdW] takes in an adjacency list representation A, as well as the lengths of the linked lists in A. It is more complicated but more efficient than the previous algorithm:

Quantum Quadrilateral Algorithm #2

1. Choose a random vertex $u \in V$, which has k neighbors $v_1, \ldots, v_k$ with adjacency lists $A[v_1], \ldots, A[v_k]$.
2. Perform Grover's search on $v_1, \ldots, v_k$ to find up to $2n^{1-c} + 1$ neighbors v_L such that $|A[v_L]| \geq n^c$ (c to be determined later). We say that these v_L have "long" lists and that all other vertices in $\{v_1, \ldots, v_k\}$ have "short" lists.
3. If there are more than $2n^{1-c}$ neighbors with long lists, then graph G must contain a quadrilateral (see explanation below). Truncate each long list to length n^c, and concatenate these truncated lists into a new list L. Perform quantum element distinctness algorithm on L to find a vertex $w \neq u$ that is adjacent to at least two neighbors v_L.
4. Otherwise, concatenate all lists (short and long) into a new list L. Perform quantum element distinctness algorithm on L to find a vertex $w \in V$ that is adjacent to at least two vertices that are adjacent to u.
5. Perform amplitude amplification on steps 1–4.

Step 1 selects a random vertex u, which has $k \leq n-1$ neighbors $v_1, \ldots, v_k$. Step 2 searches these neighbors to determine whether more than $2n^{1-c}$ of them are adjacent to at least n^c vertices (we calculate c below). This search is bounded by $O(\sqrt{n^{2-c}})$ queries.[3] Now our algorithm forks, depending on the result of the search.

If u does have more than $2n^{1-c}$ neighbors v_L with "long" lists $A[v_L]$, then G must have a quadrilateral. This is because $\sum_{v_L} |A[v_L]| > 2n^{1-c} \cdot n^c = 2n > n + k$, which means that, taken together, the lists $A[v_L]$ must contain at least two instances of at least one vertex other than u. In other words, at least one vertex $w \neq u$ must be adjacent to two vertices v_{L_1} and v_{L_2}. That w is in a quadrilateral with u, v_{L_1}, and v_{L_2}.

To find v_{L_1}, v_{L_2}, and w, we truncate $2n^{1-c} + 1$ of the long lists to length n^c and then concatenate the truncated lists. The concatenation has length $(2n^{1-c} + 1) \cdot n^c > 2n > n + k$, so it must contain at least two instances of at least one vertex $w \neq u$. We run the quantum element distinctness algorithm to find such a w and to determine v_{L_1} and v_{L_2}. Doing so requires $O(n^{2/3})$ queries and brings the query complexity to $O((n^{2-c})^{1/2} + n^{2/3})$.

[3] For some unknown t, there are t vertices $v_L \in \{v_1, \ldots, v_k\}$ such that $|A[v_L]| \geq n^c$. If $t \leq 2n^{1-c}$, then finding all v_L's (and determining that there are no more) requires $\left(\sum_{i=1}^{t} O(\sqrt{k/i})\right) + O(\sqrt{k}) = O(\sqrt{kt}) = O(\sqrt{n \cdot 2n^{1-c}}) = O(\sqrt{n^{2-c}})$ queries. If $t > 2n^{1-c}$, then we must find $2n^{1-c}+1$ of the v_L's. Doing this requires $\sum_{i=1}^{2n^{1-c}+1} O(\sqrt{k/i}) = O(\sqrt{n \cdot 2n^{1-c}}) = O(\sqrt{n^{2-c}})$ queries.

If u does not have more than $2n^{1-c}$ neighbors with long lists, then there are at most $n-1$ short lists, each with length less than n^c. In addition, there are at most $2n^{1-c}$ long lists, whose combined length is at most $2n$. We concatenate all of these lists into a new list L, whose length is bounded above by $O((n-1) \cdot n^c + 2n) = O(n^{1+c})$. Running the element distinctness algorithm on L brings our query complexity to $O\left((n^{2-c})^{1/2} + n^{2/3} + (n^{1+c})^{2/3}\right)$.

As we have seen before, these first few steps are not sufficient. If G has one quadrilateral, the probability that step 1 chooses a u in a quadrilateral is $O(1/n)$, so the overall probability of finding a quadrilateral is bounded above by $O(1/n)$. Therefore, we embed steps 1–4 in $O(\sqrt{n})$ iterations of amplitude amplification, bringing the algorithm's overall query complexity to $O\left(\sqrt{n}\left((n^{2-c})^{1/2} + n^{2/3} + (n^{1+c})^{2/3}\right)\right)$.

In order to minimize this complexity, we must find a c that minimizes the sum of $(n^{2-c})^{1/2}$ and $(n^{1+c})^{2/3}$. So, we set them equal to each other and solve for c:

$$n^{(2-c)/2} = n^{(2+2c)/3} \;\Rightarrow\; (2-c)/2 = (2+2c)/3 \;\Rightarrow\; c = 2/7.$$

Plugging $c = 2/7$ into the overall query complexity, we obtain:

$$O\left(\sqrt{n}\left((n^{12/7})^{1/2} + n^{2/3} + (n^{9/7})^{2/3}\right)\right) =$$

$$O\left(n^{1/2}(n^{6/7} + n^{2/3} + n^{6/7})\right) = O\left(n^{1/2}(n^{6/7})\right) = O(n^{19/14}).$$

This algorithm is significantly more efficient than the other algorithms in this chapter, and its efficiency is due not to a clever procedure but rather to careful balance between its search components and its built-in knowledge about when a list must contain indistinct elements. If "long" lists were defined differently, or if a different number of "long" lists were sought, this balance would be disrupted, and the efficiency would suffer.

This balance makes this algorithm unlike the other algorithms in this paper, and it deserves examination and admiration. However, its impressiveness is overshadowed by the generality of the quantum walk algorithm for arbitrary subgraphs (which of course include quadrilaterals), which is discussed in Section 5.3.

4.3 Yet Another Hypothetical Situation

What if we had a decision-only algorithm for the quadrilateral problem? Could we transform this too into an example-finding algorithm by embedding it in a recursion? By now, the answer should be obvious: We could.

As before, each recursive level would need to be repeated a sufficient number of times. Again, if the decision algorithm used an adjacency matrix, the query complexity of the example-finding algorithm would be the same as the query complexity of the decision algorithm. If it used an adjacency list, the query complexity of the example-finding algorithm would again be $O(n^2)$.

We can sense a trend in these hypothetical situations. It seems that any algorithm for determining the existence of a fixed-length cycle can be transformed into an example-finding algorithm by being embedded in a recursion, each level of which is repeated sufficiently many times. If the decision algorithm uses an adjacency matrix and no adjacency list, then the example-finding algorithm has the same asymptotic query complexity as the decision algorithm. If the decision algorithm uses an adjacency list, then the example-finding algorithm has query complexity $O(n^2)$.

4.4 Quadrilateral Lower Bounds

Like the triangle problem, the quadrilateral problem has algorithm-independent lower bounds. And as with the triangle problem, we can use the OR problem in a reduction:

Again let $X \in \{0,1\}^{\binom{n}{2}}$ be an input to the OR problem, and again think of X as a graph G on n vertices. Let the bits in X indicate which of the G's $\binom{n}{2}$ possible edges exist. Add two vertices to G, and connect them to each other and to every vertex in G. This supergraph G'' has a quadrilateral if and only if there is an edge between two of the G's original n vertices—once again, if and only if $\mathrm{OR}(X) = 1$.

So, deciding whether an $(n+2)$-vertex graph contains a quadrilateral (like deciding whether an $(n+1)$-vertex graph contains a triangle) requires at least as many queries as computing the OR of $\binom{n}{2}$ bits. Thus, classical quadrilateral algorithms are bounded below by $\Omega(n^2)$ queries, and quantum quadrilateral algorithms are bounded below by $\Omega(n)$ queries.

The above reduction gives no information about whether the triangle problem or quadrilateral problem is easier. It just says that no quadrilateral algorithm and no triangle algorithm can involve fewer than $\Omega(n^2)$ classical queries or $\Omega(n)$ quantum queries.

Chapter 5

Algorithms for Longer Cycles and Arbitrary Subgraphs

Many triangle and quadrilateral algorithms can be adapted for longer k-cycles, but the adaptations are not efficient: The classical adaptations have time complexities around $O(n^k)$, and all of the adaptations (classical and quantum) have query complexity $O(n^2)$.[1,2]

Luckily, there are better algorithms. Some of them work on cycles of any length, and some are specific to even cycles or odd cycles.

Throughout this chapter, we assume that each algorithm is devoted to k-cycles for a fixed k. In other words, we assume that there are different algorithms for different cycle lengths and that k is not inputted. Therefore, we can omit factors of k from the algorithms' complexities. And, when a complexity includes k as an exponent, we know that the exponent is fixed.[3]

[1]Unfortunately, the tactics used in Quantum Quadrilateral Algorithm #2 do not seem to apply to cycles of other lengths. The tactics could be used in a 5-cycle algorithm, but its query complexity would be $O(n^2)$. And attempts to apply the tactics to cycles of other lengths have resulted in faulty algorithms.

[2]If it was not clear before, it should now be totally clear how very uninformative query complexity is for classical algorithms.

[3]Note that we call arbitrary-length cycles *k-cycles*, even cycles *$2k$-cycles*, and odd cycles *$(2k+1)$-cycles*. These different uses for k are potentially confusing, but context should always clarify whether or not k is the cycle length under discussion.

5.1 Classical Algorithms for Longer Cycles

5.1.1 Classical Even Cycle Algorithm #1

Dana Richards and Arthur L. Liestman [RL85] showed that Classical Quadrilateral Algorithm #1 generalizes to an $O(n^k)$ time algorithm for finding $2k$-cycles. In short, the algorithm loops through ordered subsets of k vertices and looks for a unique corner between each pair of consecutive vertices. It takes in an adjacency list for G but uses it only to create another data structure:

Classical Even Cycle Algorithm #1 [RL85]

1. For each pair of vertices $u, v \in V$, use the adjacency list to find all corners—up to $2k-2$ of them—between u and v. Store the corners in a sorted list $W(u, v)$.
2. For each ordered subset $\{v_1, v_2, \ldots, v_k\}$ of k vertices in V,
3. Make temporary copies of the lists $W(v_1, v_2), W(v_2, v_3), \ldots, W(v_k, v_1)$.
4. Remove $v_1, v_2, \ldots, v_k$ from each list. (No vertex can appear twice in a cycle).
5. Truncate all lists of length greater than k to length k.
6. Create a bipartite graph $G' = (X \cup Y, E')$, where $X = \{(v_i, v_{i+1}) \mid 1 \leq i \leq k\}$, $Y = \cup_1^k W(v_i, v_{i+1})$, and $E' = \{((v_1, v_{i+1}), w) \mid w \in W(v_1, v_{i+1}), 1 \leq i \leq k\}$.[4]
7. Search for a maximal matching $M = \{((v_i, v_{i+1}), w_i') \mid 1 \leq i \leq k\}$ for G'. If M is found, then $v_1, w_1', v_2, w_2', \ldots, v_k, w_k'$ is a $2k$-cycle.

First, we must convince ourselves that the algorithm finds a $2k$-cycle if and only if one exists. The following argument follows Richards and Liestman; for the full proof, see [RL85, §4].

Suppose G contains a $2k$-cycle C^*, consisting of $v_1, x_1, v_2, x_2, \ldots, v_k, x_k$. We want to find a $2k$-cycle $v_1, w_1, v_2, w_2, \ldots, v_k, w_k$, which can be the same as C^* but does not have to be. Step 1 makes lists $W(u, v)$ of up to $2k-2$ corners between each pair of vertices in $(u, v) \in V \times V$. Step 2 chooses an ordered k-subset of V. Suppose it chooses $\{v_1, v_2, \ldots, v_k\}$, the set of $v_1, v_2, \ldots, v_k$ in C^*. Step 3 makes duplicates of the k lists $W(v_i, v_{i+1})$. (The duplicates are necessary because the original lists need to be used multiple times and cannot be altered. The duplicate lists are temporary and can be shortened as necessary.) Some of these lists have $2k-2$ elements; some have fewer.

[4]Here and everywhere in this subsection, we let v_{k+1} denote v_1.

No vertex can appear more than once in a cycle, so step 4 removes $v_1, v_2, \ldots, v_k$ from all duplicate lists. (No list $W(v_i, v_{i+1})$ contains v_i or v_{i+1}, so this pruning removes at most $k-2$ elements from each list.) Some lists may still be longer than k; step 5 truncates them to length k.

Now, let us step back from the algorithm and see that k distinct corners w_i can be each selected from the k lists $W(v_i, v_{i+1})$. Some of the lists, say j of them, began with fewer than $2k-2$ elements; call them "short" lists. The other $k-j$ lists began with $2k-2$ elements; call them "long" lists. A short list $W(v_i, v_{i+1})$ originally included every corner between v_i and v_{i+1}, including x_i. Some of the corners may have been deleted, but x_i was not. Therefore, it still includes x_i, and w_i can be set to x_i. After the deletions, a long list $W(v_i, v_{i+1})$ still has at least k corners. It might include j of the x_i's "claimed" by the short lists, but it also has at least $k-j$ "unclaimed" elements. So, we have $k-j$ long lists, each with at least $k-j$ "unclaimed" elements. This is sufficient for each remaining pair (v_i, v_{i+1}) to be assigned a distinct corner. So, there are enough corners to complete the $2k$-cycle. Here is how the algorithm makes the corner assignments:

Step 6 creates a bipartite graph G'. For ease, we will say that the graph has a "left" side and a "right" side. The vertices on left side represent the k pairs (v_i, v_{i+1}). The vertices on the right side represent the corners in $\cup_1^k W(v_i, v_{i+1})$. (Therefore, each corner is represented by just one vertex, even if it appears in multiple lists $W(v_i, v_{i+1})$.) A vertex on the left x is adjacent to a vertex on the right y if and only if y represents a corner between the pair of vertices represented by x. Step 7 runs a bipartite graph matching algorithm, which yields a complete matching and reveals the remaining vertices in the $2k$-cycle. (Whenever step 2 selects an ordered k-subset that does not generate a $2k$-cycle, step 7 does not find a complete matching.)

Now that we are convinced of the correctness of the algorithm, we can discuss its complexity. Step 1 finds up to $O(n)$ corners between each of up to $O(n^2)$ pairs, so it requires $O(n^3)$ time and $O(n^2)$ queries. Steps 3–7 run in time polynomial in k,[5] but they may need to be repeated for each of $O(n^k)$ ordered k-subsets of V. Therefore, the overall time complexity is $O(n^k)$, and the overall query complexity is $O(n^2)$.

[5]If the Hopcroft-Karp $O(n^{5/2})$ time algorithm for maximum matching [HK73] is used, steps 3–7 run in $O(k^{5/2})$ time.

5.1.2 Classical Odd Cycle Algorithm #1

Slight modifications turn Classical Even Cycle Algorithm #1 into an odd cycle algorithm [RL85]. The main difference is that step 2 loops through $O(n^{k+1})$ ordered $(k+1)$-subsets $\{v_1, v_2, \ldots, v_k, v_{k+1}\}$ such that v_{k+1} is adjacent to v_1. The rest of the algorithm is adjusted so that corners are sought for the pairs $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_{k+1})$ but not for (v_{k+1}, v_1). The modified algorithm finds a $(2k+1)$-cycle, if one exists, in $O(n^{k+1})$ time for $k \geq 2$.

5.1.3 Classical Even/Odd Cycle Algorithm #2

Thankfully, we can do much, much better. In 1985, B. Monien devised an algorithm that detects and finds k-cycles in $O((k-1)! \cdot nm) = O(nm)$ steps. Monien's trick is focusing on the nodes that paths visit, not on the order in which they visit them. In other words, his algorithm works with sets of nodes, not sequences of nodes.

However, the algorithm's lack of interest in sequence is only temporary. The eventual result of its set manipulations is the special matrix D^{k-1}, where $D^{k-1}_{i,j}$ is equal to a path of length $k-1$ from i to j if one exists, and equal to a no-path symbol λ otherwise. This matrix D^{k-1} can be compared to the regular adjacency matrix M to see if there is a $k-1$-path from i to j for any edge $(j, i) \in E$. If so, then the graph G contains a k-cycle.

However, we will not examine this algorithm here, as it is both less elegant and less efficient than the algorithm presented in the next subsection. Interested readers should consult Monien's article, [Mon85].

5.1.4 Classical Even/Odd Cycle Algorithm #3

As promised in Chapter 4, there is a k-cycle algorithm using matrix multiplication. Discovered by Noga Alon, Raphael Yuster, and Uri Zwick [AYZ95], the algorithm avoids pseudo-cycles $\langle v_0, \ldots, v_i, \ldots, v_i, \ldots v_k \rangle$ by choosing random acyclic orientations of the input graph. An *acyclic orientation* of a graph $G = (V, E)$ is a directed graph $\vec{G}$ obtained by choosing a random ordering $\prec$ of vertices and putting in $\vec{G}$ only those edges $(u, v) \in E$ such that $u \prec v$. The algorithm takes in an adjacency matrix M:

Classical Even/Odd Cycle Algorithm #3 [AYZ95]

1. Repeat until a k-cycle is found or until all acyclic orientations have been tried:
2. Choose a random ordering of the vertices in V.
3. Create an adjacency matrix $\vec{M}$ for the acyclic orientation $\vec{G}$ corresponding to the ordering.
4. Compute $\vec{M}^{k-1} \wedge M$.

A path $p = \langle v_0, v_1, \ldots, v_{k-1} \rangle$ in G might appear in $\vec{G}$ as either $\langle v_0, v_1, \ldots, v_{k-1} \rangle$ or as $\langle v_{k-1}, v_{k-2}, \ldots, v_0 \rangle$. Therefore, the probability that p appears in $\vec{G}$ is $2/(k!)$. If p is part of a k-cycle in G, we discover this with probability 1. So, if G contains a k-cycle, it appears in each $\vec{G}$ with probability $2/(k!)$. Therefore, steps 2–5 are repeated an expected number of at most $k!/2$ times.

Steps 2–5 are dominated by the calculation of $\vec{M}^{k-1}$, which requires $O(\log k)$ squarings of $\vec{M}$ and takes $O(\log k \cdot n^\alpha)$ time, where α is the exponent of matrix multiplication. Therefore, the expected time of the algorithm is $O(k! \log k \cdot n^\alpha) = O(n^\alpha)$.[6] The query complexity is, of course, $O(n^2)$.

5.1.5 Classical Even Cycle Algorithm #4

For even cycles, we can do even better. Raphael Yuster and Uri Zwick discovered that, for every $k \geq 2$, there is an example-finding algorithm for $2k$-cycles that runs in $O(n^2)$ time [YZ97]. More precisely, it runs in $O((2k)! \cdot n^2)$ time. So, if k is part of the input, the time complexity is exponential. But, since we are assuming that k is fixed, the time complexity is only $O(n^2)$. Granted, the factor of $(2k)!$ makes the running times unwieldy for large k. Nevertheless, it is astonishing that there are algorithms for finding 10-cycles, 100-cycles, and even 100,000-cycles that are asymptotically faster than classical algorithms for triangles.

The algorithm takes in an adjacency list, and the heart of the algorithm is a breadth-first search (BFS). The BFS begins at a "source" vertex s. For other $v \in V$, let $d(v)$ be the distance between v and s. Let $L_i = \{v \mid d(v) = i\}$ be the complete set of vertices at level i of the BFS tree. We know L_{i+1} by the end of stage i of the BFS. During stage i, while the set is still growing, we call it L'_{i+1}. We say that an edge is *inside* L_i if both of its vertices are in L_i, and we say that it is *between* L_i *and* L_{i+1} if one vertex is in L_i and one vertex is in L_{i+1}.

[6] Alon, Yuster, and Zwick also present a deterministic version of this algorithm [AYZ95, Section 4]. Removing the algorithm's randomness increases its time complexity to $O(n^\alpha \log n)$.

Classical Even Cycle Algorithm #4 [YZ97]

1. For each vertex $s \in V$,
2. Begin a breadth-first search from s.
3. At each stage i of the BFS, scan the linked lists of the vertices in L_i. Keep count of the number of edges found inside L_i and the number of edges found between L_i and L'_{i+1}.
4. Halt the BFS as soon as one of the following holds:
   a) Stage $k-1$ has completed or the BFS has ended;
   b) At least $4k \cdot |L_i|$ edges have been found inside L_i;
   c) At least $4k \cdot (|L_i| + |L_{i+1}|)$ edges have been found between L_i and L'_{i+1}.
5. Check whether the subgraph G' induced by BFS has a $2k$-cycle. (How to check depends on which condition caused the BFS to halt. See discussion below.)

Here is a sketch of Yuster and Zwick's proof of the algorithm's correctness and efficiency: Suppose the BFS halts because stage $k-1$ has completed (or the BFS has ended). This means that levels $L_0, L_1, \ldots, L_k$ of the BFS tree have all been finalized. Let $G' = (V', E')$ be the subgraph such that $V' = \cup_i L_i$ and E' contains all edges inside each L_i (except L_k) and between each L_i and L_{i+1}. For each level L_i, there are fewer than $4k \cdot |L_i|$ edges inside L_i, fewer than $4k \cdot (|L_{i-1}| + |L_i|)$ edges between L_{i-1} and L_i, and fewer than $4k \cdot (|L_i| + |L_{i+1}|)$ edges between L_i and L_{i+1}. (If this were not so, the BFS would have halted earlier, and under a different condition.) Therefore, $|E'| < 12k \cdot n$.

Note that G' contains all $2k$-cycles that visit s. It is easy to see this: Any cycle that starts at s and visits a vertex v in L_{k+1} needs $k+1$ edges to reach v and at least another $k+1$ edges to return to s. Therefore, any cycle that extends beyond G' has length greater than $2k$. According to Corollary 2.6 in [YZ97], we can determine whether s does in fact sit on a $2k$-cycle in $O((2k!) \cdot |V'|) = O(n)$ operations and queries.

Now suppose the BFS halts because $4k \cdot |L_i|$ edges have been found inside L_i for some $i < k$. The BFS might halt in the middle of stage i, so stage i might be incomplete. However, $L_0, L_1, \ldots, L_i$ are all finalized. Let $G' = (L_i, E')$ be the subgraph such that E' contains all edges inside L_i. Because $|E'| = 4k \cdot |L_i|$, G' must contain at least one connected subgraph $C = (V_C, E_C)$ such that $V_C \subseteq L_i$ (not necessarily including s) and $|E_C| \geq 4k \cdot V_C$. Subgraph C is either bipartite or nonbipartite. Either way, C must contain a path p of length $2j$ that can be extended into a $2k$-cycle in the BFS tree. This can be done in $O(kn) = O(n)$ operations and queries.

Finally suppose the BFS halts because $4k \cdot (|L_i| + |L_{i+1}|)$ edges have been found between L_i and L'_{i+1} for some $i < k$. An argument similar to the argument above proves that this BFS tree must also contain a $2k$-cycle, and that it can be found in $O(n)$ operations and queries.

So, in $O(n)$ operations and queries, steps 2–5 either determine that s does not appear in a $2k$-cycle or find a $2k$-cycle in G. Step 1 loops through all vertices $s \in V$, so the algorithm's overall time and query complexities are $O(n^2)$.

5.1.6 Even Cycle Theorem

Sometimes, we can know that a graph contains an even cycle of a certain length without ever running an algorithm:

Theorem 5.1 (Yuster & Zwick [YZ97]) *Let $l \geq 2$ be an integer and let $G = (V, E)$ be an undirected graph with $m \geq 100ln^{1+(1/l)}$. Then G contains a C_{2k} for every $k \in [l, lv^{1/l}]$. Furthermore, such a C_{2k} can be found in $O(kn^2)$ time. In particular, a cycle of length* exactly *$\lfloor ln^{1/l} \rfloor$ can be found in $O(n^{2+1/l})$ time.*

More simply: If a graph is sufficiently dense, it must contain even-length cycles, for lengths in a certain range.[7] For example, if $n = 10,000$ and $m = 40,000,000$ then G must contain a C_{2k} for every $k \in [4, 10]$. (Note that n must be very large for $100ln^{1+(1/l)}$ not to exceed $n(n-1)/2$, the maximum number of edges.) The proof of this theorem is beyond the scope of this paper, but it can be found in [YZ97].

5.2 Quantum Algorithms for Longer Cycles

Some of the classical algorithms for longer cycles are not conducive to quantum revision. Classical Even Cycle Algorithm #1 and Classical Odd Cycle Algorithm #1 begin by making $O(n^2)$ queries to transform the adjacency list into another data structure. We can avoid these queries by stipulating that the algorithms instead receive that data structure. Then we can replace the "for" loop in step 2 with amplitude amplification. These changes improve the algorithm's query complexity from $O(n^2)$ to $O(n^{3/2})$ for 5-cycles and 6-cycles. How-

[7] Though we never would have guessed the specifics of this theorem, we should not be surprised that such a theorem exists. Intuitively, it makes sense that a sufficiently large and dense graph contains predictable subgraphs.

ever, stipulating that the algorithms receive a non-standard data structure is an artificial fix. Furthermore, the algorithms still require $O(n^2)$ queries for cycles longer than length 6.

Amplitude amplification can also be inserted into Classical Even/Odd Cycle Algorithm #3. However, it only speeds up our selection of a good ordering and does not improve the algorithm's query complexity of $O(n^2)$ or time complexity $O(n^\alpha)$.

Still, one of the classical algorithms does have a quantum improvement.

5.2.1 Quantum Even Cycle Algorithm #1

Quantum Even Cycle Algorithm #1 is the same as Classical Even Cycle Algorithm #4, except that amplitude amplification replaces the "for" loop through vertices $s \in V$:

Quantum Even Cycle Algorithm #1

1. Choose a random vertex $s \in V$.
2. Perform steps 2–5 from Classical Even Cycle Algorithm #4.
3. Perform amplitude amplification on steps 1 and 2.

This one change improves the algorithm's query complexity from $O(n^2)$ to $O(n^{3/2})$. Perhaps the algorithm could benefit from other quantum interventions, but it does not seem likely. Instinctively, we want to replace the BFS with Grover's algorithm. However, the BFS is used here not to search but to generate a subgraph, which Grover's algorithm does not do.

5.3 Algorithms for Cycles of Arbitrary Length

In late 2003, it was shown—twice!—that a cycle of any size can be found in a subquadratic number of quantum queries. It can't be denied that these findings somewhat dim the luster of various algorithms discussed in this paper. However, working through those algorithms as we did allows us to appreciate the new findings with a sense of history, a sense of drama, and a sense of the cumulative nature of algorithmic research. Neither finding stems from an entirely new algorithm. Rather, one is a generalization of Ambainis's Algorithm, and the other is a generalization of the $O(n^{1.3})$ quantum triangle algorithm.

5.3.1 Generalization of Ambainis's Algorithm

In [CE], Andrew M. Childs and Jason M. Eisenberg show that Ambainis's $O(n^{k/(k+1)})$ algorithm for the k-element distinctness problem generalizes to a subset-finding algorithm. More

precisely, they show that Ambainis's algorithm can be repurposed to find a subset of size k that satisfies any given property P. In a graph G with n vertices, a cycle of length k is a subset of the $\binom{n}{2}$ possible edges in G. Therefore, for any k, finding a k-cycle requires $O(\binom{n}{2}^{k/(k+1)}) = O(n^{2k/(k+1)})$ queries, which is always less than $O(n^2)$ queries.

For more details, see [CE]. We don't dwell on them here, as the Childs and Eisenberg finding is slightly improved by the Magniez, Santha, and Szegedy finding described below.

5.3.2 Generalization of $O(n^{1.3})$ Triangle Algorithm

In the same paper in which they improve their quantum triangle algorithm from $\widetilde{O}(n^{1.3})$ queries to $O(n^{1.3})$ queries, Magniez, Santha, and Szegedy show that their $O(n^{1.3})$ triangle algorithm generalizes to a $O(n^{2-(2/k)})$ algorithm for finding a specified k-vertex subgraph, for $k > 3$ [MSSa]. Of course, a k-cycle is a k-vertex subgraph, so this generalization amounts to the Holy Grail of cycle algorithms.

To arrive at the $O(n^{2-(2/k)})$ bound, Magniez, Santha, and Szegedy use this parameterized query complexity:

$$O\left(r^2 + \left(\frac{n}{r}\right)^{(k-1)/2}\left(\left(\sqrt{n} \times r^{d/(d+1)}\right) + \left(\sqrt{r} \times r\right)\right)\right),$$

where d is the minimal degree of the subgraph being sought. For cycles, d is always 2, so $r^{d/(d+1)}$ is always $r^{2/3}$. In this case, instead of letting r equal $n^{2/3}$ or $n^{3/5}$ (as was done in Section 3.2.5), let $r = n^{1-(1/k)}$. Then the r^2 term dominates the expression, and the query complexity becomes $O(n^{2-(2/k)})$. For $k > 4$, this is the best complexity yet to be seen. (To facilitate comparisons between complexities, summarizing charts are included in Chapter 6.)

Chapter 6

Conclusion

6.1 What Did We Do?

When looking closely at algorithms, it can be difficult to see how they relate to each other. Therefore, these next few pages summarize the "big picture" of this paper. Side-by-side comparisons can communicate much, so: Table 6.1 reviews the elements in our quantum "bag of tricks," and Table 6.2 compares the cycle algorithms from Chapters 3, 4, and 5.

In addition to surveying algorithms, this paper examined several search-to-decision reductions. We outlined how to transform decision algorithms for element distinctness, triangle existence, and quadrilateral existence, and we saw that the resulting example-finding algorithms have the same query complexities as the original decision algorithms (provided the algorithms use adjacency matrices, not adjacency lists). Also, we posited that the same approach could transform decision algorithms for fixed-length cycles of any length, and that the resulting example-finding algorithms would also have the same query complexities as the decision algorithms.

Table 6.1: Summary of Procedures in Quantum "Bag of Tricks"

Algorithm	Query Complexity
Grover's Search Algorithm (for unique solution in N bits)	$O(\sqrt{N})$
Grover's Search Algorithm (for 1 of t solutions)	$O(\sqrt{N/t})$
Grover's Search Algorithm (for all t solutions)	$O(\sqrt{tN})$
Amplitude Amplification (where a is prob. of "good" observation)	$O(1/\sqrt{a})$
Ambainis's Algorithm (for element distinctness)	$O(N^{2/3})$
Ambainis's Algorithm (for element k-distinctness)	$O(N^{k/(k+1)})$

Table 6.2: Summary of Cycle Algorithms and Complexities

Algorithm	Adj. Matrix or Adj. List?	Query Complexity	Time Complexity
Note: In each category, the most efficient algorithm is in boldface.			
Triangle Algorithms			
Classical Triangle Algorithm #1	Matrix	$O(n^2)$	$O(n^3)$
Classical Triangle Algorithm #2	List	$O(n+m)$	$O(n^3)$
Classical Triangle Algorithm #3*	Matrix	$O(n^2)$	$O(n^\alpha)$
Quantum Triangle Algorithm #1	Matrix	$O(n^{3/2})$	-
Quantum Triangle Algorithm #2	Matrix	$O(n+\sqrt{nm})$	-
Quantum Triangle Algorithm #3	List	$O(n^{3/2})$	-
Quantum Triangle Algorithm #4	Matrix	$\widetilde{O}(n^{10/7})$	-
Quantum Triangle Algorithm #5	Matrix	$\widetilde{O}(n^{1.3})$	-
Quantum Triangle Algorithm #5I**	Matrix	$O(n^{1.3})$	-
Classical Quadrilateral Algorithm #1	List	$O(n+m)$	$O(n^2)$
Classical Quadrilateral Algorithm #1$M^\dagger$	Matrix	$O(n^2)$	$O(n^2)$
Quantum Quadrilateral Algorithm #1	Matrix	$O(n^{3/2})$	-
Quantum Quadrilateral Algorithm #2	List	$O(n^{19/14})$	-
Classical Even ($2k$) Cycle Algorithm #1	List	$O(n^2)$	$O(n^k)$
Classical Odd ($2k+1$) Cycle Algorithm #1	List	$O(n^2)$	$O(n^{k+1})$
Classical Even/Odd (k) Cycle Algorithm #2	Matrix	$O(n^2)$	$O(nm)$
Classical Even/Odd (k) Cycle Algorithm #3*	Matrix	$O(n^2)$	$O(n^\alpha)$
Classical Even ($2k$) Cycle Algorithm #4	List	$O(n^2)$	$O(n^2)$
Quantum 5-Cycle Algorithm #1‡	Other‡	$O(n^{3/2})$	-
Quantum 6-Cycle Algorithm #1‡	Other‡	$O(n^{3/2})$	-
Quantum Even ($2k$) Cycle Algorithm #1	List	$O(n^{3/2})$	-
Generalization of Ambainis's Algorithm	Matrix	$O(n^{2k/(k+1)})$	-
Generalization of $O(n^{1.3})$ Triangle Alg.	Matrix	$O(n^{2-(2/k)})$	-

* *In the time complexity, α is the exponent of matrix multiplication.*

** *See Footnote 6 in Chapter 3.*

† *See Footnote 2 in Chapter 4.*

‡ *For a description of this algorithm's data structure, see Sections 5.1.1 and 5.2.*

6.2 What's Left?

Of the algorithms presented here for the first time, only Quantum Quadrilateral Algorithm #2 "rebalances" the relative importances of the search steps and the non-search steps. This rebalancing tactic should be kept in mind as researchers continue on their mission to unify the lower and upper bounds of cycle-finding problems.

Bibliography

[AAKV01] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. In *Proceedings of STOC '01*, pages 50–59, 2001. quant-ph/0012090.

[Amba] Andris Ambainis. Quantum lower bounds for collision and element distinctness with small range. quant-ph/0305179, 29 May 2003.

[Ambb] Andris Ambainis. Quantum walk algorithm for element distinctness. quant-ph/0311001, 1 Nov 2003.

[ASE91] Noga Alon, Joel H. Spencer, and Paul Erdös. *The Probabilistic Method*. Wiley, New York, 1991.

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

[AYZ97] Noga Alon, Raphy Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[BBC+98] Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald de Wolf. Quantum lower bounds by polynomials. In *39th IEEE Symposium on Foundations of Computer Science*, pages 352–361, 1998. quant-ph/9802049.

[BBHT98] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(4–5):493–505, 1998.

[BDH+01] Harry Buhrman, Christoph Dürr, Mark Heiligman, Peter Høyer, Frédéric Magniez, Miklos Santha, and Ronald de Wolf. Quantum algorithms for element distinctness. In *Proc. 16th IEEE Conference on Computational Complexity*, pages 131–137, 2001. quant-ph/0007016.

[BdW] Harry Buhrman and Ronald de Wolf. Private e-mail communications.

[BHMT02] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. In *Quantum Computation and Information*, number 305 in Contemporary Mathematics. American Mathematical Society, 2002. quant-ph/0005055.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[CE] Andrew M. Childs and Jason M. Eisenberg. Quantum algorithms for subset finding. quant-ph/0311038, 6 Nov 2003.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[DJ92] David Deutsch and Richard Jozsa. Rapid solutions of problems by quantum computation. In *Proceedings of the Royal Society of London, Series A*, volume 439, pages 553–558, 1992.

[dW01] Ronald de Wolf. *Quantum Computing and Communication Complexity*. PhD thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2001.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 212–219, 1996. quant-ph/9605043.

[HK73] J. E. Hopcroft and R. M. Karp. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[Kem03] J. Kempe. Quantum random walks: An introductory overview. *Contemporary Physics*, 44:307–327, 2003. quant-ph/0303081.

[Mon85] B. Monien. How to find long paths efficiently. In *Analysis and Design of Algorithms for Combinatorial Problems*, number 25 in Annals of Discrete Mathematics, pages 239–254. Elsevier, 1985.

[MSSa] Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum algorithms for the triangle problem. quant-ph/0310134 version 2, 7 Nov 2003.

[MSSb] Frédéric Magniez, Miklos Santha, and Mario Szegedy. An $\widetilde{O}(n^{1/3})$ quantum algorithm for the triangle problem. quant-ph/0310134 version 1, 21 Oct 2003.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, Cambridge, UK, 2000.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley Longman, Reading, MA, 1994.

[RL85] Dana Richards and Arthur L. Liestman. Finding cycles of a given length. In *Cycles in Graphs*, number 27 in Annals of Discrete Mathematics, pages 249–255. Elsevier, 1985.

[Sho97] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[Sim97] Daniel Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.

[Sze] Mario Szegedy. On the quantum query complexity of detecting triangles in graphs. quant-ph/0310107, 16 Oct 2003.

[vL90] J. van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 525–631. Elsevier, Amsterdam, The Netherlands, 1990.

[YZ97] Raphael Yuster and Uri Zwick. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics*, 10(2):209–222, 1997.