

High-Level Quantum Programming with Quantum Walks

by

Héctor J. García

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Sciences
(Computer and Information Science)
at the The University of Michigan-Dearborn
2007

Master Thesis Committee:

Professor Igor L. Markov, Chair
Professor David Yoon
Professor Michael Lachance
Professor Bruce Elenbogen

© Héctor J. García 2007
All Rights Reserved

To my father, José A. Ojeda-Santini, who, by all mundane standards, was to be known as my step-father.
For teaching me that, while intelligence is the path to knowledge, it is hard work that spawns true wisdom.
And to my mother, Elsie M. Ramírez, who always supports my dreams with love and solidarity.

ACKNOWLEDGEMENTS

None of this would have been possible without the love and support of my wife – Gracias, Vida. Many thanks to Prof. Igor L. Markov for taking me on as a student when he had no obligation to do so. Thanks to Prof. Lachance and Dr. George F. Viamontes for useful discussions.

ABSTRACT

High-Level Quantum Programming with Quantum Walks

by

Héctor J. García

Quantum Walks, the quantum analogs of classical stochastic walks, have been identified by several groups as a potential framework for the development of new quantum algorithms. Several known quantum algorithms, including Grover's quantum search algorithm, have already been significantly generalized using quantum-walk architectures. Furthermore, recent studies have identified some surprising features of quantum walks that may be exploited for algorithmic purposes including partial decoherence.

Simulation of quantum walks has been limited to independent numerical studies, and no general simulation environment tool exists. In this work, we extend the high-performance quantum computing simulation tool *QuIDDPro* from the University of Michigan. We implement new functionality that enables ease of experimentation of quantum walk dynamics.

TABLE OF CONTENTS

DEDICATION	ii
ACKNOWLEDGEMENTS	iii
LIST OF FIGURES	iii
LIST OF TABLES	iv
CHAPTER	
I. Introduction	1
1.1 Motivation	2
1.2 Objectives	4
1.3 Organization	5
II. Survey of Quantum Computation	6
2.1 Quantum Mechanics	6
2.1.1 Quantum Mechanics Basics and the Dirac Notation	7
2.1.2 Measurements and Decoherence	9
2.2 Computing with Quantum Systems	10
2.2.1 Quantum Registers	11
2.2.2 Quantum Gates and Circuits	11
2.2.3 Example of a Quantum Algorithm	14
III. Overview of Discrete-Time Quantum Walks	16
3.1 Preliminary Definitions	16
3.2 Quantum-Walk Algorithm	18
3.3 Simple Quantum-Walk Example	21
3.4 Decoherence Effects in Discrete Quantum Walks	23
IV. QuWalkLib – High-Level Quantum Programming Library for Quantum Walks	26
4.1 Low-Level Primitives	27
4.2 High-Level Data Structure	28
4.3 Quantum Register Manipulation	28
4.4 Quantum Circuit (Operator) Manipulation	31
4.5 Simulation of Quantum Walks	33
V. Design of Efficient Quantum-Walk Circuits (QWCs)	39

5.1	One-Dimensional QWC	39
5.2	Two-Dimensional QWC	47
5.3	Speed-ups of the Quantum Walk	48
VI. Future Work: Simulation-based		
	Algorithm Design with Quantum Walks	50
6.1	Quantum-Walk Solution to the Heat Equation	50
6.1.1	Definition of the Heat-Equation Problem	51
6.1.2	Classical Solution using Random Walks	53
6.1.3	Quantum-Walk Formulation of the Heat-Equation Problem	55
6.1.4	Measurement and Extraction of the Solution	56
6.2	Improving High-Level Quantum Programming	57
APPENDICES		59
BIBLIOGRAPHY		69

LIST OF FIGURES

Figure

2.1	Computational basis states for a single qubit.	10
2.2	Graphical representation of the quantum circuit {NOT, CNOT}.	13
3.1	Movement of a particle in a random walk.	17
3.2	Square structure for 2-qubit graph register.	21
3.3	Quantum Circuit for walk on a square.	22
3.4	Complete mathematical evolution of the quantum walk on a square.	23
3.5	Quantum walk on a square with decoherence of the graph space on every iteration.	24
4.1	Architectural overview of QuWalkLib.	27
4.2	Probability distributions of vertices in a 5-qubit cycle quantum walk after 20 iterations. The distribution on the left was not initialized to a symmetrical superposition, thus yielding an asymmetrical distribution.	37
4.3	Comparison of the probability distribution of a classical and quantum walk on a line (the classical case is plotted in blue), after 100 iterations. This graph was proposed by Kendon and Tregenna in [20].	38
5.1	6-qubit Binary Incrementer Quantum Circuit.	41
5.2	Simulation of the decrementer circuit using the incrementer circuit.	42
5.3	Simulation of C^m -NOT gates using $4(m - 2)$ Toffoli gates and $m - 2$ work qubits. The first group of qubits identified by m are the controlling qubits while the group of qubits identified by $m - 2$ are the work qubits. Note that the C^m -NOT gate is simulated correctly without pre-initialized work qubits. Also note that, after applying all the Toffoli gates, the original state of the work qubits is preserved.	44
5.4	Simulation of the C^8 -NOT gate with two C^4 -NOT and one ancilla qubit. Note that the ancilla qubit must be set to $ 0\rangle$ prior to the operation and will be reset to that same value upon termination of the operation.	46
6.1	Steady-state temperature distribution for 10 x 30 plate with 100° heat applied on the top edge and 0° applied on the remaining three edges.	54

LIST OF TABLES

Table

2.1	Graphical Representation of Basic Quantum Gates	13
4.1	Sample Low-Level Primitives Executed by QuIDDPo	27
4.2	QRegister Class Sample Functions	28
4.3	QPCircuit Class Sample Functions	31
4.4	QWalk Class Sample Functions	33

CHAPTER I

Introduction

As computer technology keeps evolving and manufacturers continue designing ever smaller circuits, it is evident that *quantum-mechanical* effects will start playing a major role in the development of computer algorithms. It comes as no surprise then that *Quantum Computation and Information Science* has received a lot of attention from several research communities. The speed-ups offered by Shor's quantum factoring algorithm [28] and Grover's quantum search algorithm [13] over their corresponding classical formulations helped fuel interests and expectations about the possibilities of exploiting quantum-mechanical effects to improve computational tasks. However, in recent years, the field has experienced a relative lull in the development of new algorithms due, in part, to the difficulty in understanding the counter-intuitive behavior of quantum effects. The work in [29] outlines the current state of the field and offers insights into the causes of this scarcity in new quantum algorithms. Despite an apparent lack of breakthrough progress, several incremental results are showing a lot of promise to harness quantum behavior for development of new algorithms. Among these, *quantum walks* offer some indication of doing for quantum algorithms what classical randomized algorithms have done for their deterministic counterparts.

1.1 Motivation

Quantum walks are the quantum analogs of classical random (stochastic) walks. The usual interpretation of a random walk in classical dynamics –that of a particle positioned on a vertex of a graph that chooses to move to an adjacent vertex with some probability– applies to the quantum case. However, quantum walks exhibit markedly different behavior than their classical counterparts due to the properties of closed quantum systems. It is through the algorithmic manipulation of these effects that we wish to attain new computational speed-ups unavailable in the classical case.

Quantum walks in continuous time were first studied by Aharonov, Davidovich and Zanguri [3] in the Physics community. Discrete-time quantum walks were later introduced to the Computer Science community by Meyer [21] and Watrous [32]. Since then, a number of findings identifying and analyzing quantum effects on graph structures have appeared with some interesting and surprising results. Ambainis [4] formulated the element distinctiveness problem in terms of quantum walks, achieving polynomial speed-up over the classical case. It was shown in [2] that the mixing time, the time it takes the walk to reach a stationary distribution, is also polynomially smaller in the quantum setting. Kempe [17] proved that it is exponentially faster to quantum walk a hypercube from one corner to the opposite corner (hitting time). Finally, Shenvi, Kempe and Whaley [27] have generalized Grover’s quantum search algorithm based on a quantum walk architecture, offering improved flexibility in how the search is implemented.

It is well known that quantum walks exhibit the cited speed advantages due to the phenomenon of *quantum interference*, which allows paths that have the same final destination to reinforce or cancel each other out. What is less known and quite surprising is that *quantum decoherence* can play an important part too. Decoherence is a phenomenon that

occurs when a quantum system comes in contact with the environment. In simple terms, it causes parts of a quantum system to “collapse” to a classical system. Since quantum systems are very sensitive to decoherence effects, which can also be thought of as noise, designing scalable quantum circuits that maintain coherence has been a major challenge for the engineering community. However, Kendon and Tregenna [20] have shown that partial decoherence in a quantum walk can actually enhance the quantum speed-ups to a certain degree, in stark contrast to intuition. This is encouraging for the engineering of more practical and reliable quantum algorithms. That is, if we can implement a particular quantum algorithm (such as quantum search) using a quantum-walk architecture, it should in theory make it less vulnerable to noise than a non-quantum-walk implementation, given that the quantum walk can endure a certain degree of decoherence while still achieving the desired speed-up. The latter is still an open question which, like most other challenges related to quantum walks, requires a great deal of novel analysis and, in particular, simulation. Kendon [19] has noted the relative difficulty in obtaining tight bounds analytically for certain properties of quantum walks, compared to numerical simulation. Furthermore, independent simulations of quantum walks have yielded improved quantum-walk algorithms. One example is Shenvi’s [26] discovery of a quantum-walk structure by numerical simulation that allowed Ambainis, Kempe and Rivosh [5] to develop a grid search algorithm with polynomial speed-up. As another example, Kempe’s proven exponentially faster-than-classical hitting time algorithm was simultaneously found numerically by Yamasaki [33]. The value of empirical analysis and simulation seems even greater when we consider recent findings by Flitney, Abbott and Johnson [10] regarding another quantum-walk property, *history dependence*, and the role it plays in the new emerging field of *Quantum Game Theory*. History dependence in random walks is the mechanism by which each step of the walk depends on prior steps. In the quantum setting, the effects of history

dependence are augmented by interference and local reversibility, making quantum walks extremely sensitive to this memory effect and exhibiting behavior similar to Parrondo's paradox in Game Theory¹ [11]. This has led to the formulation of other game-theoretic results in the quantum context using quantum walks. See [12] for a nice survey on Quantum Game Theory oriented towards economists and game theorists. On the other hand, only a few of these formulations have been effectively simulated by a handful of researchers, limiting the exposure of Quantum Game Theory to a small audience. Providing a helpful and comprehensive software package that simulates quantum-walk structures would encourage other research communities, like economists, mathematical modelers and game theorists, to look deeper into Quantum Game Theory and the prospects of quantum walks for the development of new algorithms.

1.2 Objectives

Despite these important contributions and strong potential to re-energize future research in quantum algorithms, simulation of quantum walks has been limited to independent studies, and no general simulation environment tool exists for implementing quantum walks on different graph structures with measurable decoherence. Consequently, the main objective of this thesis paper is to provide a high-level quantum programming framework that facilitates implementation of quantum-walk simulations. The main advantages of providing a quantum-walk simulation framework are: (i) enable users to easily experiment with quantum walks on several types of graphs and the quantum dynamics involved while allowing empirical validation of these concepts, and (ii) explore partial decoherence effects on quantum walks to understand its effects on quantum speed-ups, as well as obtain numerical results comparable to existing analytical results. This high-level quantum programming language will leverage and expand on previous work by the *University of*

¹In Game Theory, Parrondo's paradox arises when a combination of two losing games result in a winning game.

Michigan's Quantum Circuits Group [15] (QCG) and their exceptional `QuIDDPro` software.

`QuIDDPro` is a high-performance quantum-circuit simulation software developed by Viamontes, Markov and Hayes [31] at QCG. `QuIDDPro` uses a clever data structure called *QuIDD –Quantum Information Decision Diagram–* developed by the same authors. It allows `QuIDDPro` to take advantage of the repetitive structural composition in various quantum computing operators. For certain quantum computations, like Grover's search algorithm, `QuIDDPro` performs significantly faster than other generic simulators, as illustrated in [31]. In our research, we expand the current capabilities of `QuIDDPro` and include high-level functionalities that allow the following: (i) setup and implementation of quantum-walk structures, and (ii) partial decoherence simulation functions.

The final objective of this research paper is to present a practical application of the high-level quantum programming framework described above. Specifically, we simulate a quantum walk on a lattice structure to try and obtain a numerical solution to the heat equation on two dimensions. As will be shown later in this thesis, the simulation of quantum walks with a degree of decoherence holds promise for performing the above tasks efficiently, although considerable research still needs to be conducted to make this claim.

1.3 Organization

The rest of this thesis is organized as follows. Chapter II will provide a brief survey of quantum computation to prepare the reader for the topics covered in the rest of the paper. Chapter III reviews some important concepts of quantum walks and their properties. Our `QuIDDPro`-based high-level quantum programming language is presented in Chapter IV. Finally, Chapter V looks at future research work in quantum programming and algorithm design with quantum walks.

CHAPTER II

Survey of Quantum Computation

In general, computational devices rely on physical phenomena to carry out computations. The behavior of digital computers, for example, conforms to the theoretical framework of electromagnetism. Quantum computers, however, carry out computations with quantum systems and hence are subject to quantum-mechanical laws. In order to understand the dynamics involved in quantum computing and quantum walks, we first describe the concepts of Quantum Mechanics that are most relevant. The purpose of this chapter is to provide a brief survey to highlight these concepts.

In the rest of this paper, the terms *classical computer* or *classical computation* will refer to devices that are governed by the classical theories of physics, such as digital and analog computers.

2.1 Quantum Mechanics

When Quantum Mechanics (QM) was originally developed in the 1920's, it presented a new paradigm in the way we look at the world. Since then, QM has evolved into the most robust theoretical framework in modern science. This paper cannot provide an in-depth review of QM. Instead, we focus the discussion on the mathematical concepts that are relevant to understanding Quantum Computation.

Perhaps the most non-intuitive fact behind Quantum Mechanics is the notion that a sub-

atomic particle can also behave like a wave. This principle, known as the wave-particle duality, has outstanding implications in how we evaluate the dynamics of a particular quantum system. For instance, in a classical system, there is a linear amount of particle *observables* (position, momentum, etc.) that one needs to keep track of in order to model the system. On the other hand, the wave-like properties of particles in a quantum system makes the task of modeling such a system exceedingly difficult and complex because we need to keep track of an exponential number of quantities that are related to the probability distribution of particle observables, not their definite values. Therefore, in QM we use a notational device, known as *Dirac Notation* that helps to concisely represent the complex dynamics of a quantum system. This notation takes its name from its original creator, renowned physicist Paul Dirac, and is also known as *bra-ket* notation.

2.1.1 Quantum Mechanics Basics and the Dirac Notation

We now review the basics of Dirac notation. We will describe each relevant mathematical concept using Dirac notation. Later in the chapter, we will see how these concepts are used in Quantum Computation. The reader is assumed to have some basic knowledge of Linear Algebra.

State Vectors

In Dirac notation, we represent the state of a quantum system (the collection of possible observable quantities) using a *ket* $|\psi\rangle$. A ket is nothing but a vector of values. In our case, those values happen to be complex ones. For example, a random ket would look something like $|\psi\rangle = \begin{pmatrix} \text{i} \\ \sqrt{2} \end{pmatrix}$, where $\text{i} = \sqrt{-1}$. Thus, since we are using complex vectors to model the system, we say that the system lives in linear *Hilbert* space \mathcal{H}_n , where n is the size of the set of orthonormal basis vectors¹ that span \mathcal{H}_n . This set of vectors is also known as the

¹The set of vectors V is said to be orthonormal if each $v_i \in V$ is both orthogonal (inner product between v_i and v_j is 0) and normal (norm of v_i is 1).

basis states. The fact that it is linear means that when we manipulate these kets, adding or multiplying them, the resulting ket also lives in the same space. This is also known as an *inner product* space (the term “inner product” will be explicitly defined later on). Lets take the sample ket shown above. We can have this particular ket live in \mathcal{H}_2 space, which we say is spanned by orthonormal basis vectors, $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Using the labels 0 and 1 we can write the Dirac notation equivalent of the two basis vectors as $|0\rangle$ and $|1\rangle$, respectively. Having defined the basis states of the system, we are now able to write an arbitrary state vector in \mathcal{H}_2 as a linear combination of basis states $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ with the added restriction that $|\alpha|^2 + |\beta|^2 = 1$, which is a consequence of the fact that quantum systems undergo unitary evolution. We can now decompose our sample ket like so, $|\psi\rangle = i \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \sqrt{2} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = i |0\rangle + \sqrt{2} |1\rangle$.

Given that a ket $|\psi\rangle$ is a complex vector, we can write its complex-conjugate transpose as $\langle\psi|$, and we call it a *bra*. For example, we can calculate the bra of our sample ket to get, $\langle\psi| = (-i \quad \sqrt{2})$.

The *inner product* of a bra $\langle\psi|$ and a ket $|\varphi\rangle$ is known as a *bra-ket*, and is denoted, $\langle\psi|\varphi\rangle$. Basically, the inner products gives us the probability of obtaining vector state $|\psi\rangle$ from the current vector state $|\varphi\rangle$ upon measurement of $|\psi\rangle$. We will discuss the details on how measurements affect a state vector in the next section.

Finally, the *tensor product* or *Kronecker product* operation between kets or bras allows us to combine smaller Hilbert spaces into larger ones. For instance, using the basis states defined in the example above, $|0\rangle \otimes |1\rangle = |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$.

The concepts described above are only a small subset of the vast theory that governs the quantum mechanical world. For a deep and comprehensive look at the subject, we refer the reader to [1].

Operators

Knowing the basics of how to describe a quantum system, we can now discuss some details as to how to manipulate such systems. This task is performed by means of quantum operators. Let $|\psi_i\rangle$ be a state vector with some particular value at time i , then operator O will act on $|\psi_i\rangle$ and map the current state to $|\psi_{i+1}\rangle$, where $|\psi_{i+1}\rangle$ is a new state determined by the mathematical structure of O . This is concisely expressed in the following way, $|\psi_i\rangle \xrightarrow{O} |\psi_{i+1}\rangle$ or $O|\psi_i\rangle = |\psi_{i+1}\rangle$. Mathematically, quantum operators are square matrices with dimensions that are equivalent to those of the state vector on which they act. There are a couple of other properties that must hold for any quantum matrix operator O ,

- (i) *Linearity* - If $|\psi_i\rangle$ lives in Hilbert space \mathcal{H}_n , then after applying O , $|\psi_{i+1}\rangle$ will also live in that same space.
- (ii) *Unitarity* - applying O will not change the norm of the vector state, which is equal to 1.

There are special kind of operators which are related to the observable quantities in a quantum system. These will be discussed in the next section.

2.1.2 Measurements and Decoherence

A measurement is the basic mechanism which allows extraction of information from a closed quantum system. Measurements, therefore, “open” the quantum system and cause it to decohere (collapse), either partially or completely, into a classical system with some probability. In order to measure a quantum system, we make use of what are known as *projective-measurement* operators. These particular operators have the property that when they are applied to a quantum system, their eigenvalues correspond to the possible outcomes of the measurement. Formally, if a quantum system is in state $|\psi\rangle$, then applying projective-measurement operator \hat{O} will yield measurement outcome m with probability,

$$(2.1) \quad p_m = \langle \psi | \hat{O} | \psi \rangle$$

If we obtain outcome m after applying \hat{O} , then the new state of the quantum system is,

$$(2.2) \quad \frac{\hat{O} | \psi \rangle}{\sqrt{p_m}}$$

Projective measurements are not the only types of measurements that we make on a quantum system. However, for the purposes of this paper, projective measurements will suffice. We refer the reader to [1] for a survey on the subject. Now that we have some notion of the basics of QM and Dirac notation, we can focus our discussion on Quantum Computation.

2.2 Computing with Quantum Systems

Just like in classical computation, we define a basic unit of information for Quantum Computation. We call such a unit the qubit. As opposed to a classical bit, which is represented by a simple binary state, the qubit is represented by a ket vector. Following the discussion in Section 2.1.1, we know that these qubit vectors live in complex Hilbert space since the underlying system behaves quantum mechanically. Furthermore, we use the labels used in classical computation, namely, 0 and 1, as the two possible values that can be measured on a qubit (eigenstates). We now have a quantum computational basis denoted by vectors $|0\rangle$ and $|1\rangle$ for the case of a single qubit of information. The vector representation of the basis states was first shown in Section 2.1.1 and is shown again in Figure 2.1 for completion.

$$|0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Figure 2.1: Computational basis states for a single qubit.

Thus, as described in Section 2.1.1, a qubit can exist in a superposition of both 0 and 1. In Section 2.2.2, we will see how we can achieve such a superposition through the use of quantum operators.

2.2.1 Quantum Registers

Computing with a single qubit is not very interesting. Hence, we want to expand our range of computational basis states by adding additional qubits to the system. We do so by means of the tensor product. Formally, we define an n -qubit quantum register as the state vector that is formed by “tensoring” (Kronecker product) n single qubit state spaces \mathcal{H}_2 (see Section 2.1.1), which is denoted by $\mathcal{H}_2^{\otimes n}$. Quantum registers serve the same purpose as a normal processor register with the added functionality that, because of linear superposition, they can represent exponentially many more states than classical registers.

2.2.2 Quantum Gates and Circuits

Quantum gates are essentially quantum operators. Following the discussion in Section 2.1.1, we define a quantum gate acting on k -qubits as a $2^k \times 2^k$ unitary matrix that performs a certain transformation on the system. When gates are applied to a qubit system, the state vector of the system is multiplied by the gate’s matrix in order to obtain the new state vector. Since the gate is a unitary matrix, this implies that the computation is reversible—their is a one-to-one correspondence between the inputs and outputs. Similar to classical computation, there is a universal set of elementary gates that has been proven to satisfy all our computational needs [9]. We describe some of these basic gates bellow.

NOT - This is a 1-qubit gate that “flips” or negates the current value of the qubit. The

NOT gate makes the transformations $NOT |0\rangle = |1\rangle$ and $NOT |1\rangle = |0\rangle$.

$$NOT = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

CNOT - This is a 2-qubit gate that negates the second qubit if the first qubit is set to $|1\rangle$.

The CNOT gate makes the transformations $CNOT|10\rangle = |11\rangle$ and $CNOT|11\rangle = |10\rangle$. Any other input states are left unchanged.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Toffoli - This is a 3-qubit gate that negates the third qubit if the first and second qubits are set to $|1\rangle$. The Toffoli gate makes the transformations $TOFF|110\rangle = |111\rangle$ and $TOFF|111\rangle = |110\rangle$. Any other input states are left unchanged.

$$TOFF = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Hadamard - This is a 1-qubit gate that creates an unbiased superposition of states $|0\rangle$ and $|1\rangle$ as described in Sections 2.1.1 and 2.2. The Hadamard gate makes the transformations $HAD|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $HAD|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Note that we can apply the Hadamard gate to larger qubit spaces by “tensoring” single qubit Hadamard gates together.

$$HAD = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$





Using the small set of quantum gates we have defined we can create arrays that can be applied to a quantum system sequentially. Such an array of quantum gates is also known as a quantum circuit. For example, if we have a 2-qubit system and apply a *NOT* gate and later a *CNOT* gate, then our quantum circuit is $\{\text{NOT}, \text{CNOT}\}$. Hence, quantum gates

are the building blocks of quantum circuits. Most of the discussions in Chapter V will revolve around designing efficient quantum circuits to perform quantum walks in one and two dimensions.

Graphical Representation

To facilitate the description of quantum circuits, we use the customary graphical representation that is used in the field of Quantum Computation. Table 2.1 shows what those graphics look like for the gates previously described.

Table 2.1: Graphical Representation of Basic Quantum Gates

<i>Gate</i>	<i>Graphical Symbol</i>
NOT	
CNOT	
Toffoli	
Hadamard	

The horizontal lines seen in the drawings are called wires. They represent the passage through time of the qubit system and are read from left to right. For example, a graphical representation for the sample quantum circuit mentioned in the previous section would look like the graphic shown below.

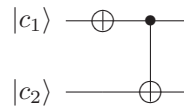


Figure 2.2: Graphical representation of the quantum circuit {NOT, CNOT}.

Note that wires are usually labeled from top to bottom. Vertical lines indicate which ones are the controlling and target qubits in a gate. The “•” symbol indicates that the gate is activated when the state of the controlling qubit is $|1\rangle$. If a “○” (not shown in the figure) symbol is used instead of a “•” the gate is activated if the state of the controlling qubit is

$|0\rangle$. The “ \oplus ” symbol indicates which target qubits are negated when the gate is activated. Finally, other gates like the Hadamard gate are depicted using a labeled box that covers the wires on which they act.

2.2.3 Example of a Quantum Algorithm

Having discussed the basic concepts of Quantum Computation, we might wonder how they all come together to carry out a specific algorithm. We now show an example of how a quantum algorithm works. In particular, we take a look at the quantum search algorithm, which achieves a quadratic speed-up over its classical counterpart.

Grover’s Quantum Search

The quantum search algorithm was originally developed by L. K. Grover [13]. Algorithm 1 shows pseudo-code for a quantum search on a system that lives in \mathcal{H}_N space. The computational basis states in \mathcal{H}_N correspond to the indices of the elements inside an unsorted database. We want to query the database in order to find a particular index x . Note that we only need $\log_2(N) = n$ qubits to index the entire set of elements in the database. Furthermore, two quantum registers are used in the algorithm, one is the index register and the other is used as temporary workspace.

Algorithm 1 Quantum Search

```

1: initialize system
2:  $i \leftarrow 0$ 
3:  $r \leftarrow \frac{\pi\sqrt{N}}{4}$ 
4: while  $i < r$  do
5:   apply Grover operator  $G$ 
6:    $i \leftarrow i + 1$ 
7: end while
8: measure

```

The key step in the algorithm is the application of the Grover operator G , which we describe using four main instructions:

Apply oracle operator O - The oracle operator is a quantum circuit that implements a

function $f(x)$, where x is an arbitrary computational basis state of the index register. The result of applying function $f(x)$ is either 1, if x turns out to be the index we are looking for, and 0 otherwise. Formally, the oracle operator O performs the transformation $|x\rangle|0\rangle \xrightarrow{O} |x\rangle|f(x)\rangle$, where the additional qubit stores the result of applying $f(x)$.

Apply Hadamard gate of size n - The Hadamard gate $H^{\otimes n}$ is applied to all the qubits in the index register to reach an equal superposition of all possible states. With the oracle function applied, the state of the system is now $\sum_{x=0}^{N-1} (-1)^{f(x)} |x\rangle$. Essentially, the solution has now been marked by the oracle function with a phase shift (the state is set to $-|x\rangle$).

Perform conditional phase shift - This is a special operator that increases the probability of obtaining the marked solution upon measurement of the system.

Looking at Algorithm 1, we can see that the Grover operator is applied $\frac{\pi\sqrt{N}}{4}$ times in order to reach a probability close to 1 of measuring the marked solution. This makes the asymptotic complexity of the algorithm $O(\sqrt{N})$, which is a quadratic speed-up over the classical search algorithm. The specific mathematical details of how the algorithm works are beyond the scope of our discussion. However, in Chapter III we will look at another quantum algorithm, namely, the quantum walk algorithm, in full detail.

CHAPTER III

Overview of Discrete-Time Quantum Walks

Quantum walks come in two flavors –continuous and discrete. There are important differences between these two types of quantum walks. The work in [5] contains concise descriptions and comparisons. Given the subject of this paper, we will restrict ourselves to the discrete-time formulation only, since it shows the most algorithmic potential. Moreover, it has been shown by Childs et al. [8] that we can always use a discrete-time formulation to effectively simulate the continuous-time walk.

3.1 Preliminary Definitions

Discrete quantum walks are defined on a graph $G(V, E)$ where V is the set of vertices and E is the set of edges. Each edge connects adjacent vertices. This is similar to the classical formulation of random walks. The dynamics of the walk are also formulated in similar fashion with a “particle” starting at a particular vertex and moving to either of its adjacent vertices with some probability. Figure 3.1 shows five iterations of a classical random walk with the corresponding accumulated probabilities. Note that after several iterations, the probability distribution of the vertices starts resembling the Gaussian bell-curve.

The important difference between the classical and quantum case is that the graph G is modeled by complex vectors in Hilbert space \mathcal{H}_N and spanned by orthonormal basis states $|v\rangle$ (see Section 2.1.1), where N is the size of the Hilbert space. We highlight the

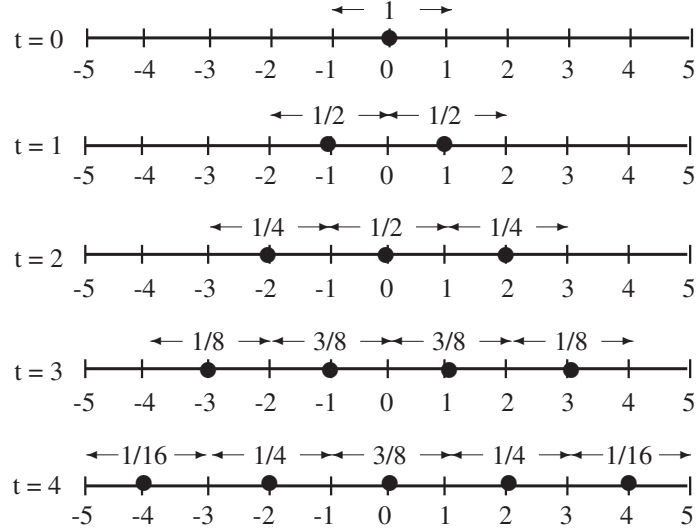


Figure 3.1: Movement of a particle in a random walk.

following properties about this structure,

- (i) $v \in V$,
- (ii) the set V is indexed by $k \in \mathbb{Z}$,
- (iii) N is the total number of vertices v in G .

Also, G is a d -regular graph with each vertex $|v\rangle$ having a set of labeled edges $\{e_v^j \in E \mid j = 1, 2, \dots, d\}$, such that e_v^j is the j^{th} edge leaving vertex $|v\rangle$ and ending in vertex $|\tilde{v}_j\rangle$. If $d = N$ then we have a complete graph. Once the regular-graph model has been defined, it is simple to modify it for irregular sparse graphs. We will postpone this definition until the next section.

In order to introduce edge weights, we augment \mathcal{H}_N with the so-called *coin space* \mathcal{H}_d spanned by the states $|1\rangle, |2\rangle, \dots, |d\rangle$, which are the possible states of the quantum coin. For example, in the cases where we use an unbiased 2-sided coin, then we set $d = 2$, which can be used to assign a weight of $1/2$ to two adjacent edges. Note the use of the same integer-index scheme for coin states as that used for edges. Ideally, we want the total number of possible states of the coin to equal the regular-edge degree of the graph, but

this is not necessarily so, and often leads to implementation issues for large graphs. The total Hilbert space of the quantum system would then be $\mathcal{H}_{qw} = \mathcal{H}_N \otimes \mathcal{H}_d$ and the wave function reads:

$$(3.1) \quad |\psi\rangle = \sum_{j=1}^d \sum_{v=1}^N a(e_v^j, v) |\tilde{v}_j\rangle \otimes |v\rangle$$

where $a(e_v^j, v)$ is a complex number.

3.2 Quantum-Walk Algorithm

The general algorithm that implements a quantum walk can be described by a small set of instructions:

Algorithm 2 Quantum Walk

- 1: initialize system
 - 2: **for** each iteration of the walk **do**
 - 3: flip coin
 - 4: shift position
 - 5: **end for**
 - 6: measure
-

We now review each step of Algorithm 2 in detail.

Step 1: Initialization As shown in [19] and [30], the initial state of the quantum walk can have diverse effects on its evolution. In particular, the walk will evolve asymmetrically with a drift towards the state in which it started. This effect is a result of the local reversibility property of quantum systems. To correct this effect, we initialize the coin space to a symmetric superposition that keeps half the evolution in the Real domain and half in the Imaginary domain. Suppose the possible states of the coin operator are $|1\rangle, |2\rangle, \dots, |j\rangle, \dots, |d\rangle$, then we obtain a symmetrical superposition by applying a *Discrete Fourier Transform* (DFT) unitary operator of size d ,

$$(3.2) \quad C_{DFT}^d = \frac{1}{\sqrt{d}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & & w^{d-1} \\ 1 & w^2 & \ddots & & w^{2(d-1)} \\ \vdots & & & \ddots & \vdots \\ 1 & w^{d-1} & w^{2(d-1)} & \dots & w^{(d-1)(d-1)} \end{bmatrix}$$

where w is the d^{th} root of unity $e^{2\pi i/d}$. Consequently, we define the origin of the walk to be at vertex $|0\rangle$, and apply operator C_{DFT}^d once to get the following system state,

$$|\psi_{coin}\rangle = \frac{1}{\sqrt{d}} \sum_{j=1}^d e^{2j\pi i/d} |j\rangle \otimes |0\rangle$$

Steps 2 – 5: Iterative Walk The walk itself is a step iteration of two instructions. We can think of a particle that starts at the origin, and on each iteration moves (takes a step) to an adjacent vertex with some probability based on the weight of the edge connecting them.

flip coin - this instruction is performed by applying a unitary operator C on \mathcal{H}_d .

For example, if we wish for the particle to move to any of its adjacent vertices with equal probability, then C is a *Hadamard* operator $H^{\otimes d}$. Note that if we use a symmetrical operator (such as that of Equation 3.2) as the coin-flip operator, then there is no need to initialize the coin space, and we can skip Step 1 of the quantum-walk algorithm.

shift position - this instruction controls the particle displacement to adjacent vertices. Define a shift operator S on \mathcal{H}_d that maps the state $S|j, v\rangle$ to $|j, \tilde{v}\rangle$ for the corresponding edge e_v^j that connects vertex v to its j^{th} neighbor \tilde{v} .

Now that we have defined the shift operator S , it is easy to modify the general model described in Section 3.1 to simulate sparse irregular graphs. All that is needed is to

keep d as the maximal vertex degree, and make S a conditional operator such that,

$$(3.3) \quad S |j, v\rangle \longrightarrow \begin{cases} |j, \tilde{v}\rangle & \text{if } e_v^j = (v, \tilde{v}) \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

What we are doing, in practice, is keeping the regularity of the graph but adding self-loops to vertices with degree less than the maximal degree d . These self-loops will prevent the walker from advancing to the non-adjacent vertices upon application of the shift operator. Although we are describing the iterative part of the walk as two separate instructions, in reality, the operators representing these instructions can be synthesized using several algorithms [24] to produce a new unitary operator $U = S \cdot (C \otimes I)$, where I is the identity operator. This implies that we can generalize the evolution of the walk, and describe it concisely as,

$$(3.4) \quad |\psi_i\rangle = U^i |\psi_0\rangle$$

Step 6: Measurement The final step is to measure the system after t iterations of the walk. Although we choose to present it here as a final step, the role of measurement in a quantum-walk process is far from trivial, and we could very well choose to perform partial measurements at precise moments in the walk. We may choose to measure the coin space, the position space or both at some chosen time (iteration) t . For example, upon measurement of system $|\psi_t\rangle$ on position basis state $|v\rangle$, the system will be found in said state with probability,

$$(3.5) \quad p_v = \sum_{j=1}^d |\langle v, j | \psi_t \rangle|^2$$

We shed some light on the topic of measurement and decoherence in Section 3.4.

3.3 Simple Quantum-Walk Example

In general, quantum walks require at least 2 composite quantum qubit systems, which we call registers. The first register represents the coin space while the second represents the graph space. The simplest quantum walk possible uses 1 qubit for each of the registers. However, the behavior of the walk on a structure like this would be trivial and fail to demonstrate interference effects.

We now look at an example of a quantum-walk structure composed of a 1-qubit coin-space register and a 2-qubit graph-space register. Given the size $2^2 = 4$ of the graph register we can implement a walk on a square with each vertex labeled by a permutation of the possible states of the register as seen in Figure 3.2.

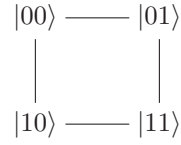


Figure 3.2: Square structure for 2-qubit graph register.

For this example, since we have a small number of qubits, we will use a Gray-code¹ representation of the vertices of the graph rather than the integer-labeling scheme defined in Section 3.1. Furthermore, the small graph space means that we don't have to worry about initializing the system since the walk will not have enough steps to evolve asymmetrically. Following Section 3.2, we now look at the coin-flip and shift-position operators. Since we are using a 1-qubit coin register and the graph is 2-regular, we can use a *Hadamard* gate C_H to perform an unbiased coin-flip operation,

¹Binary Gray Code: 0 = 00, 1 = 01, 2 = 11, 3 = 10

$$(3.6) \quad C_H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The shift operator S will be a circuit composed of two CNOT gates which will modify the graph register according to the state of the coin,

$$(3.7) \quad \begin{aligned} S |0\rangle \otimes |i, j\rangle &\longrightarrow |0\rangle \otimes |i, 0 \oplus j\rangle & i, j = 0, 1 \\ S |1\rangle \otimes |i, j\rangle &\longrightarrow |1\rangle \otimes |1 \oplus i, j\rangle \end{aligned}$$

Figure 3.3 shows a graphical representation of the operators defined in Equations 3.6 and 3.7. $|\psi_i\rangle$ is the state of the whole system at iteration i . $|c\rangle$ is the coin register, and $|p_1\rangle \otimes |p_2\rangle$ span the graph register.

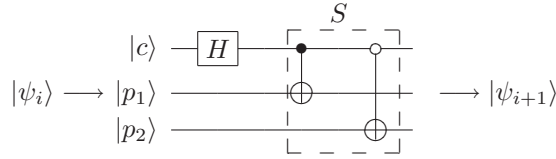


Figure 3.3: Quantum Circuit for walk on a square.

The mathematical calculation of the evolution of the system is depicted in Figure 3.4, and the `QuIDDP` script that implements this walk is shown in Appendix A. Running the simulation script and displaying the vector state on each iteration will produce the same results as the mathematical evolution shown in Figure 3.4. It is worthwhile to note the effects of destructive interference in iterations 2, 3, 6, and 7. The walk also shows a periodic property by returning to its initial configuration ($|000\rangle$) after only 8 iterations. Following our discussion from Section 3.2, we perform a single measurement after all iterations of the walk in order to maximize the quantum behavior (superposition, interference) of the walk. The resulting state after the single measurement will be $|000\rangle$ with a probability equal to 1.

$$\begin{aligned}
|000\rangle &= |\psi_0\rangle \\
|\psi_0\rangle &\xrightarrow{H} \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|001\rangle + |110\rangle) = |\psi_1\rangle \\
|\psi_1\rangle &\xrightarrow{H} \frac{1}{2}(|001\rangle + |101\rangle + |010\rangle - |110\rangle) \xrightarrow{S} \frac{1}{2}(|000\rangle + |011\rangle - |100\rangle + |111\rangle) = |\psi_2\rangle \\
|\psi_2\rangle &\xrightarrow{H} \frac{1}{2\sqrt{2}}(2|011\rangle + 2|100\rangle) = \frac{1}{\sqrt{2}}(|011\rangle + |100\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|10\rangle = |\psi_3\rangle \\
|\psi_3\rangle &\xrightarrow{H} \frac{1}{2}(|010\rangle + |110\rangle + |010\rangle - |110\rangle) = |010\rangle \xrightarrow{S} |011\rangle = |\psi_4\rangle \\
|\psi_4\rangle &\xrightarrow{H} \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|010\rangle + |101\rangle) = |\psi_5\rangle \\
|\psi_5\rangle &\xrightarrow{H} \frac{1}{2}(|010\rangle + |110\rangle + |001\rangle - |101\rangle) \xrightarrow{S} \frac{1}{2}(|000\rangle + |011\rangle + |100\rangle - |111\rangle) = |\psi_6\rangle \\
|\psi_6\rangle &\xrightarrow{H} \frac{1}{2\sqrt{2}}(2|111\rangle + 2|000\rangle) = \frac{1}{\sqrt{2}}(|111\rangle + |000\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|01\rangle = |\psi_7\rangle \\
|\psi_7\rangle &\xrightarrow{H} \frac{1}{2}(|001\rangle - |101\rangle + |001\rangle + |101\rangle) = |001\rangle \xrightarrow{S} |000\rangle = |\psi_8\rangle
\end{aligned}$$

Figure 3.4: Complete mathematical evolution of the quantum walk on a square.

As seen in this simple example, a quantum walk behaves drastically different than a classical random walk. We will see more complicated examples in Chapters IV and VI.

3.4 Decoherence Effects in Discrete Quantum Walks

In a quantum walk, we can model decoherence (see Section 2.1.2) by modifying Equation 3.4 to read,

$$(3.8) \quad |\psi_{t+1}\rangle = (1-p)U|\psi_t\rangle + pM_bU|\psi_t\rangle$$

where p is the probability of applying a measurement (decoherence) per time step and M_b is the measurement operator. In the rest of this paper, we use the terms decoherence and measurement interchangeably.

There are several degrees of freedom on how to apply measurements and introduce decoherence. For our purposes, we are interested in determining whether the walker has reached a particular vertex. We call such a vertex a “boundary” and say that the walker

is “absorbed” upon reaching the chosen boundary. How do we know that a walker has reached a boundary in a quantum system? We can only accomplish this by “opening” the quantum system and measuring the graph space. Mathematically, we define an absorbing boundary on vertex $|b\rangle$ by applying measurement operator M_b to $|\psi_t\rangle$. If the result of the measurement operation is the state $|b\rangle$, the walker has reached the boundary. However, as presented in Equation 3.5, even if the walker in fact reached state $|b\rangle$, there is only a probability of detecting the absorption. We will see some practical implications of this behavior in Chapter VI where we research a possible application for quantum walks. If measurements are postponed until all iterations of the walk have been completed ($p = 0$ in Equation 3.8), the quantum properties of the walk will be maximized, but the amount of information that we can extract from it will be limited (including whether a walker has been absorbed). The other extreme is to perform measurements and decohere the graph space after every iteration of the walk ($p = 1$ in Equation 3.8). This will tell us whether a walker has been absorbed on every iteration, but will cause the quantum walk to reduce to a classical random walk. To provide a clearer view of this concept. Let us look at how the evolution depicted in Figure 3.4 will look if we introduce graph-space (qubits p_1 and p_2) measurements into every iteration of the walk. Figure 3.5 shows the results of the modified walk for the first couple of iterations. Appendix B shows the additional `QuIDDP` commands needed to simulate the behavior in Figure 3.5.

$$\begin{aligned}
|000\rangle &= |\psi_0\rangle \\
|\psi_0\rangle &\xrightarrow{H} \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|001\rangle + |110\rangle) \xrightarrow{M_b} |001\rangle = |\psi_1\rangle \\
|\psi_1\rangle &\xrightarrow{H} \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle) \xrightarrow{S} \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle) \xrightarrow{M_b} |000\rangle = |\psi_2\rangle
\end{aligned}$$

Upon measuring the first iteration, we get state $|001\rangle$ with probability $|1/\sqrt{2}|^2 = .5$. Note that we could have gotten state $|110\rangle$ with the same probability. The same follows for all iterations.

Figure 3.5: Quantum walk on a square with decoherence of the graph space on every iteration.

These facts imply that some compromise must be made between quantum efficiency and information extraction in order to obtain practical results from a quantum walk. Fortunately, it has been shown in [20] that the introduction of a small amount of decoherence can actually enhance the quantum properties of the walk. The possible algorithmic implications of using decoherence in a quantum walk will be discussed in Chapter VI.

CHAPTER IV

QuWalkLib – High-Level Quantum Programming Library for Quantum Walks

Having discussed the most important concepts of quantum computation in Chapter II, we now present a C++ implementation of a high-level quantum programming language, which we call `QuWalkLib`. The relevant work on quantum walks discussed in Chapter III allows us to ground our discussion by focusing on those aspects of the implementation that are relevant to the simulation of quantum walks. The rest of the discussion in this chapter will assume basic knowledge of C++ programming concepts and techniques.

As briefly stated in Section 1.2, the programming work developed for this paper is based on previous work presented in [31]. `QuWalkLib` is an external library of C++ classes that link to the `QuIDDP` static library. In terms of architecture, Figure IV shows a graphical representation of how `QuWalkLib` interacts with `QuIDDP`.

`QuWalkLib` uses a standard template library `stringstream` class interface to send appropriately parsed commands to the `QuIDDP` static library. The `QuIDDP` runtime library gets initialized upon execution in order to process the parsed commands. The result is returned in the form of a “QuIDD wrapper” interface object (`Quiddpro::Qw`). Although this approach might seem inefficient because of the additional command-parsing job, note that the execution time of the parsing job is asymptotically smaller than the execution time related to the manipulation of QuIDD data structures that happens behind

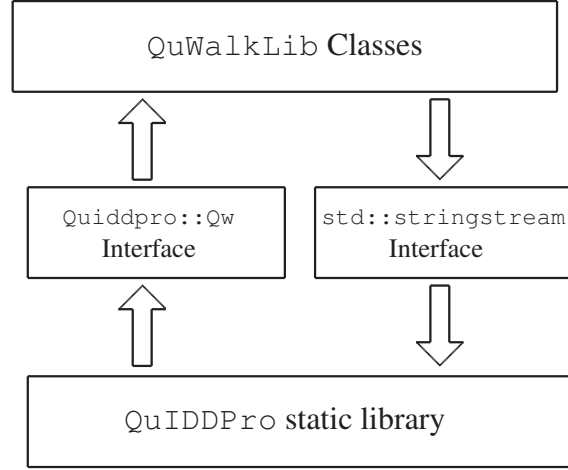


Figure 4.1: Architectural overview of QuWalkLib.

the scenes. Thus, the overall performance of the application is not affected.

4.1 Low-Level Primitives

We define low-level primitives as the linear-algebraic concepts (defined in Chapter II) that are used to manipulate qubits. For the most part, the QuIDDPPro runtime library takes care of processing and executing these instructions while the role of QuWalkLib is limited to only parsing and declaring the appropriate commands. Table 4.1 contains a sample set of operations that can be executed by QuIDDPPro. A complete list can be found in [31].

Table 4.1: Sample Low-Level Primitives Executed by QuIDDPPro

<i>Low-level Primitive</i>	<i>QuIDDPPro Functions</i>	<i>Reference</i>
Create computational basis state vector	<code>cb()</code>	p.7
Create Hadamard gate	<code>hadamard()</code>	p.18 p.21
Create custom controlled gate	<code>cu_gate()</code>	p.18 p.21
Calculate Kronecker product of vectors	<code>kron()</code>	p.7
Calculate conjugate transpose of vector	<code>conj()</code>	p.7

However, when constructing high-level quantum programs that require classical post-processing of measured results, the capabilities of QuIDDPPro are somewhat limited. Hence QuWalkLib implements a high-level data structure that performs two main tasks, (i) allows flexibility to construct hybrid quantum-classical C++ programs, and (ii) hides

the details related to low-level primitive functions so the programmer can focus on high-level procedures.

4.2 High-Level Data Structure

The library provides the end-user with a flexible C++ Application Programming Interface (API) for construction and manipulation of quantum high-level data structures such as the composite quantum systems or registers that were discussed in Section 3.3. Specifically, the `QRegister` C++ class in `QuWalkLib` provides such a high-level structure. Table 4.2 shows a series of sample prototype functions for this class.

Table 4.2: QRegister Class Sample Functions

	<i>Prototype</i>	<i>Reference</i>
Constructor function that initializes the register	<code>QRegister(string, unsigned int);</code>	p.28
Access a sub-register	<code>QRegister get_subreg(unsigned int);</code>	p.28
Adds a qubit “wire” to the register	<code>void add_wire(bool);</code>	p.28
Appends a register	<code>void append(QRegister);</code>	p.28
Applies a circuit to the register	<code>void apply_circ(QPCircuit);</code>	p.31
Returns the probability distribution for all possible states of the register	<code>vector<double> get_prob_dist();</code>	
Measures the register	<code>bool measure(string)</code>	p.28

The main advantage in using high-level structure manipulation in a quantum programming language, like the one just defined, is that it allows development of quantum-primitive-based subroutines that can be reused without the need to worry about low-level qubit addressing.

4.3 Quantum Register Manipulation

The `QRegister` class contains several native methods that allow manipulation of a quantum register without the need to explicitly construct and apply the required quantum operators. We now show some of these methods while demonstrating examples of how to use the `QRegister` class.

Register Initialization Quantum registers need to be initialized to an element of the computational basis. This action is performed in the following sample code.

```
■ QPRegister areg("areg", "000");
```

initializes a 3-qubit register to $|000\rangle$
and assigns it to QuIDDPPro variable
“areg”

```
■ areg.set_st(3);
```

sets the state of the register to $|3\rangle =$
 $|011\rangle$

As seen in the code above, initialization of a quantum register can be done upon declaration of a QPRegister object using the constructor, or by using the `void QPRegister::set_st(std::string)` method after the object has been declared. If the user needs to increase the size of the register, the following code may be used,

```
■ areg.add_wire(true);
```

adds a qubit wire to register “areg”,
the added wire is set to $|1\rangle$

Combining Registers Design of complex quantum algorithms usually requires the use of several quantum registers, as opposed to a single one. Therefore, QuWalkLib provides the user with a convenient way of combining sub-registers to form larger registers.

```
■ areg << breg;
```

combines registers “areg” and
“breg”, “areg” is now the root
register

```
■ breg.append(areg);
```

another form of combining registers,
“breg” is now the root register

The advantage of this functionality is that it allows the user to maintain a single root register while still having access to the relative qubit addresses that compose the

sub-registers. Another great advantage will become apparent when we discuss the manipulation of operators in Section 4.4.

Register Measurements The only way to extract information from the quantum register is to perform a measurement. The result of the measurement can then be read by a classical algorithm for post-processing. The user has a couple of options for performing measurements.

```
■ bool result = areg.measure("000");
```

measures areg using a projective measurement on computational basis state $|000\rangle$

```
■ bool result = areg.measure(2);
```

measure the qubit with indexed by 2

Note that both methods return a boolean value. In the case of the first method shown, we get a probabilistic measurement on the register. That is, the result will return `true` if the outcome of the measurement was the same as the binary string argument, otherwise it returns `false`. In the second example, we are measuring a single qubit. The result will be `true` if the measurement on the qubit turned out to be 1, and `false` if it turned out to be 0. Both methods leave the original state of the register intact. To obtain the correct post-measurement state of the system (see Section 2.1.2) the user needs to explicitly modify the register using the appropriate operator.

Obtaining the Probability Distribution of States The user can look at the probability distribution of all the possible states of the register with the command shown below.

```
■ vector<double> pdist = areg.get_pdist();
```

get a probability distribution of states

This is specially useful in the simulation of quantum walks where we want to take snapshots of the probability distribution of the vertices at a particular point in time. However, the user needs to be mindful of the fact that the size of the vector returned is 2^n , where n is the size of the register.

4.4 Quantum Circuit (Operator) Manipulation

The second high-level structure that has been implemented as part of the `QuWalkLib` library is the `QPCircuit` class. This data structure maintains an array of quantum gates which the programmer can manipulate easily. Table 4.3 shows some of the prototype functions of the class.

Table 4.3: QPCircuit Class Sample Functions

	<i>Prototype</i>	<i>Reference</i>
Constructor function that initializes the circuit	<code>QPCircuit(unsigned int);</code>	p.31
Adds a new gate to the circuit	<code>void add_gate(string);</code>	p.11 p.31
Removes a gate from the circuit	<code>void rem_gate(unsigned int);</code>	
Inserts a gate into the circuit	<code>void ins_gate(unsigned int);</code>	p.31
Resets the circuit in the <code>QuIDDP</code> run-time	<code>void reset();</code>	
Gets the name of the next gate in the circuit	<code>string get_next_gate();</code>	

What makes this data structure particularly useful is that complex circuits can be designed by using a single class method (see below). Moreover, the circuit maintains a relative qubit-addressing scheme. In other words, when applied to a sub-register, the circuit calculates the absolute qubit addresses that should be used when applying the low-level operators for each gate in the circuit. This makes circuit objects completely reusable so that they can be applied to different registers or sub-registers without the need to re-create each gate operator. Examples are shown below.

Circuit Initialization The circuit needs to be initialized with the number of wires (qubits) on which it will act.

■ <code>QPCircuit acirc(5);</code>	initializes a 5-qubit circuit containing 0 gates
------------------------------------	---

Adding a Gate Gates can either be added to the circuit (the gate will be appended at the end) or inserted into the circuit at a particular position provided that it exists. The parameter to both methods is a string that contains an encoding for the gate. The encoding used is the same as the one shown in [25]. Basically, gates are encoded like a function where the letter determines the gate type (T = Toffoli, C = CNOT, N = NOT, etc.), and the numbers in the argument determine the controlling (if applicable) and target qubits.

■ <code>circ.add_gate("T(1,2;3)");</code>	adds a Toffoli gate with control qubits 1 and 2, and target qubit 3
---	--

■ <code>circ.ins_gate("N(3)", 1);</code>	inserts a NOT gate with target qubit 3 at position 1
--	---

Applying a Circuit to a Register Circuits are applied to registers by calling the `apply_circ` method from the register class and passing the circuit as a parameter. An example is shown below using a register object called “areg”.

■ <code>areg.apply_circ(acirc);</code>	applies circuit “acirc” to register in “areg”
--	--

Also, we can apply the same circuit to two different sub-registers. If we have a root register called “rootreg” with two sub-registers each of them containing five qubits. Then we can apply “acirc” to both sub-registers as shown below.

■ <code>QPRegister sreg = areg.get_subreg(0); sreg.apply_circ(acirc);</code>	applies circuit “acirc” to sub-register 0 in “areg”
--	--

```

■ QPRegister sreg =          applies circuit “acirc” to sub-register
  areg.get_subreg(1);        1 in “areg”
  sreg.apply_circ(acirc);

```

4.5 Simulation of Quantum Walks

The final high-level structure that is implemented in `QuWalkLib` is based on the two structures previously described. The `QPWalk` class can be used to simulate one-dimensional and two-dimensional quantum walks. It handles all the details related to the creation of proper registers and application of necessary operators in a quantum-walk simulation. The functions in the `QPWalk` class reduce an entire quantum-walk simulation to only a few lines of code. Several prototype functions are shown in Table 4.4.

Table 4.4: QPWalk Class Sample Functions

	<i>Prototype</i>	<i>Reference</i>
Constructor function that initializes the walk	<code>QPWalk(bool, unsigned int);</code>	p.18
Runs the walk for a number of iterations	<code>void run(unsigned int);</code>	p.18
Initializes the walk	<code>void init();</code>	p.18
Resets the walk to start at the vertex of origin	<code>void reset();</code>	
Gets the coin register	<code>QPRegister get_coinreg();</code>	p.16
Gets the x-dimension graph register	<code>QPRegister get_xdimreg();</code>	p.18
Gets the circuit that is applied to the coin register	<code>QPCircuit get_coinop();</code>	p.18

Initializing the Walk The quantum walk can be initialized in the constructor method when the object is declared or later in the code to re-run the walk. These two methods are presented below.

```

■ QPWalk mywalk(true, 16);    initializes the 2-D quantum walk ob-
                               ject with 16 vertices

```

```

■ mywalk.init();              re-initializes the quantum walk ob-
                               ject

```

Running the Walk To run the walk, the programmer needs to call a single method

and pass the number of iterations as an argument. When all iterations of the walk are done, the programmer can access each of the individual register objects (coin, graph), and use any of the native methods from the `QRegister` class for those objects.

■ <code>mywalk.run(100);</code>	runs 100 iterations of the quantum walk
■ <code>mywalk.get_xdimreg();</code>	returns the x -dimension graph register

Simulating Decoherence `QuWalkLib` provides a convenient way of simulating decoherence in a quantum walk along the lines of the concepts seen in Section 3.4. This is done by means of a custom random-number generator class called `QPRandom`. Whenever the `QWalk` class is initialized we also initialize `QPRandom` with a particular seed. The method `double QPRandom::random()` is used to obtain a pseudo-random number that is used as needed in the `QWalk` class. Hence, we can easily model 3.8 inside the `QWalk` class.

■ <code>mywalk.set_decoh_factor(.5);</code>	sets the decoherence factor $p = .5$
---	--------------------------------------

Only the most important aspects of `QuWalkLib` have been reviewed in this chapter. For a complete description of `QuWalkLib` classes and functions see [16]. In order to test the functionality of all the classes described above, let's look at an implementation of the quantum walk on a cycle using `QuWalkLib`.

Quantum Walk on a Cycle

We extend the original quantum-walk example shown in Section 3.3 as a starting point in our implementation of the quantum walk on a cycle.

The first modification to our quantum-walk example is to increase the number of graph-space qubits in the model. Adding just 3 qubits will increase the amount of vertices N in the graph from $2^2 = 4$ to $2^5 = 32$. Now that we have more vertices, we can upgrade the graph structure from a square to a cycle. We will keep the same 1-qubit coin-flip operator C_H as in the original example since a cycle is also a 2-regular graph (we can only chose to move left or right). Finally, we will use integer labeling for vertices as defined in Section 3.1. This allows us to concisely define the new shift operator S ,

$$(4.1) \quad \begin{aligned} S |\leftarrow\rangle \otimes |i\rangle &\longrightarrow |\leftarrow\rangle \otimes |i+1\rangle & i \in \mathbb{Z} \\ S |\rightarrow\rangle \otimes |i\rangle &\longrightarrow |\rightarrow\rangle \otimes |i-1\rangle \end{aligned}$$

It is easy to see that S is simply a binary incrementer/decrementer operator since i is the integer value of a computational basis binary string. Furthermore, since our graph is a cycle, S should also make the following mappings,

$$\begin{aligned} S |\leftarrow\rangle \otimes |N-1\rangle &\longrightarrow |\leftarrow\rangle \otimes |0\rangle \\ S |\rightarrow\rangle \otimes |0\rangle &\longrightarrow |\rightarrow\rangle \otimes |N-1\rangle \end{aligned}$$

Note that Equation 4.1 also works for implementing a quantum walk on a line. The only difference is that, on a line, the shift described above is not performed. Rather, the walk is stopped if either of the boundary vertices, $|0\rangle$ or $|N\rangle$, are reached.

Note that our position-shift circuit is now considerably more complex, as compared to the walk on a square. Hence, we need to start considering efficiency issues in the implementation of our circuits. The topic of quantum circuit synthesis is very important in the development of current and new quantum algorithms. Large quantum gates (gates involving more than three qubits) have been shown, in practice, to be exceedingly difficult to implement with current technologies. Thus, we want to simulate these large quantum

gates with a proven universal set of smaller sized elementary gates (see Section 2.2.2). We will touch upon this subject when we discuss specific quantum-walk circuit design examples in Chapter V. See [24] for a comprehensive review of the topic.

A Note about Symmetry in Quantum Walks

Since the quantum walk on the cycle just defined requires a higher number of vertices, we need to properly initialize the coin space in order to obtain a symmetrical walk (see Section 3.2). We use a degree-2 DFT operator,

$$C_{DFT}^2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & i \\ i & 1 \end{bmatrix}$$

that initializes our coin space to a symmetrical superposition $(|\leftarrow\rangle + i|\rightarrow\rangle)/\sqrt{2}$, where \leftarrow and \rightarrow are just the labels that we have chosen to represent the possible left and right states of the coin (we could very well have chosen integers as done in the original example). Note that operator C_{DFT}^2 should not be confused with the actual coin-flip operator C_H . The degree-2 DFT operator is only applied once, while operator C_H from Equation 3.6 is applied on every iteration of the walk.

The probability distributions of the vertices of the graph after 20 iterations of the asymmetrical and symmetrical walks are shown in Figure 4.2. Note that we have plotted the data for the even numbered vertices since the odd vertices all have probability zero just like in the classical case (the opposite would be true if we had plotted an odd number of iterations).

The `QuWalkLib` code that simulates the quantum walk described above is shown in Appendix C.

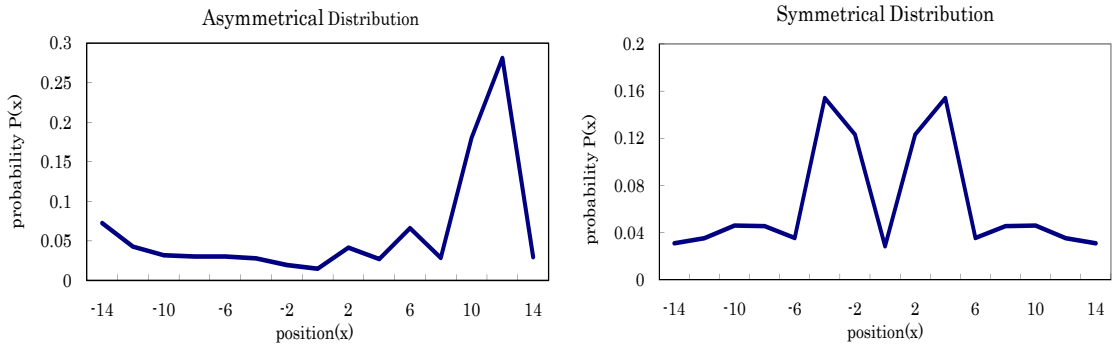


Figure 4.2: Probability distributions of vertices in a 5-qubit cycle quantum walk after 20 iterations. The distribution on the left was not initialized to a symmetrical superposition, thus yielding an asymmetrical distribution.

A Note about the Probability Distribution of Quantum Walks

There is an additional interesting fact about Figure 4.2, note the stark difference between the graphs in the figure and those described for the classical random walk in Section 3.1. The classical case follows a Gaussian (bell curve) distribution with the limiting vertices having an exponentially low probability of being reached by the walker. In the quantum case, the complete opposite happens. As the number of iterations increases, the limiting vertices have a much higher probability of being reached than vertices that are closer to the vertex of origin. Figure 4.3 shows a comparison between the quantum and the classical case.

The consequences of this quantum behavior in the context of developing new quantum algorithms is still open for debate in the Computer Science community. We shed some light on this topic in Chapter VI and discuss possibilities for future work in this area.

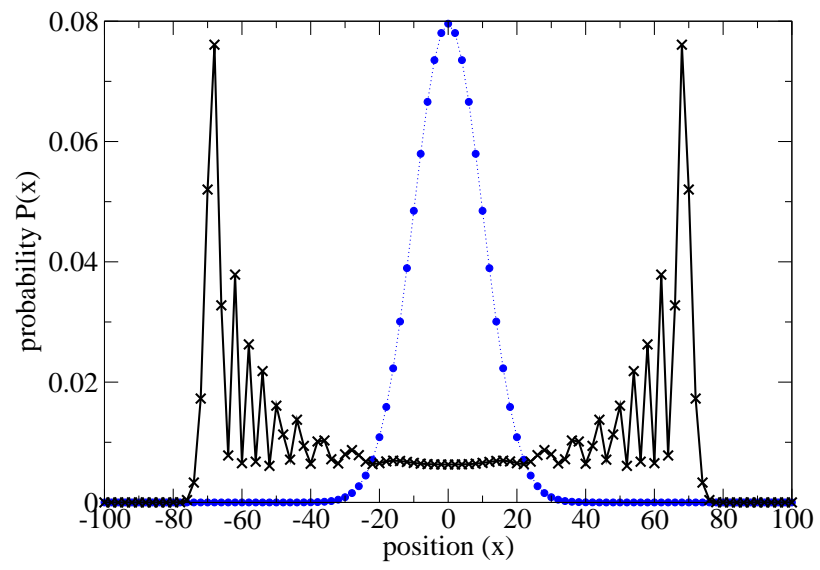


Figure 4.3: Comparison of the probability distribution of a classical and quantum walk on a line (the classical case is plotted in blue), after 100 iterations. This graph was proposed by Kendon and Tregenna in [20].

CHAPTER V

Design of Efficient Quantum-Walk Circuits (QWCs)

Since the overall performance of a quantum walk is dependent on the efficiency of the underlying circuit, careful consideration should be given to reducing gate count and unnecessary ancilla qubits that can undermine its performance. In this chapter, we discuss several designs of quantum circuits for performing quantum walks in one and two dimensions. We also calculate their overall complexity and efficiency.

5.1 One-Dimensional QWC

Following the discussion of Section 4.5, we know that in order to implement the one-dimensional quantum walk on a line or cycle we need a quantum binary incrementer circuit that acts on the graph-space register to map $|i\rangle$ to $|i + 1\rangle$, where i is a binary number representing the vertex label. This mapping will simulate the motion of the particle from its current vertex position on the line to its adjacent vertex on the right as discussed in Section 3.1 and shown in Figure 3.1.

In general, for a binary number x_n composed of the vector of bits $\{b_n b_{n-1} \dots b_0 \mid b_j \in \{0, 1\}\}$ of size n , the increment operation is performed by flipping (negating) bit b_j iff all previous bits $b_{j-1} b_{j-2} \dots b_0$ are set to 1. After this, we clear all bits in $b_{j-1} b_{j-2} \dots b_0$ by setting them to 0. Algorithm 3 provides a concise illustration of this procedure. For example, say bits $b_{k-1} b_{k-2} \dots b_0$ in x_n are all set to 1 and therefore bit b_k needs to be flipped, upon

execution of Algorithm 3 we can expect four main results,

Algorithm 3 Binary Increment

Require: Bit vector x_n with value d

Ensure: Bit vector x_n with value $d + 1$

```

1: for  $i = 0$  to  $n$  do
2:   if  $x_n[i]$  is false then
3:      $x_n[i] \leftarrow \text{true}$ 
4:     if  $i > 0$  then
5:       for  $j = i - 1$  to  $0$  do
6:          $x_n[j] \leftarrow \text{false}$ 
7:       end for
8:     end if
9:     return  $x_n$ 
10:  else if  $i$  equals  $n$  then
11:    for  $h = n$  to  $0$  do
12:       $x_n[h] \leftarrow \text{false}$ 
13:    end for
14:    return  $x_n$ 
15:  else
16:    do nothing
17:  end if
18: end for

```

- (i) the bit b_k is flipped (negated),
- (ii) all bits b_i , where $i < k$, are all flipped,
- (iii) all bits b_i , where $i > k$, are not modified,
- (iv) if all n bits are set to 1, they are all reset to 0.

Algorithm 3 can be simulated in the quantum context with the use of multi-controlled NOT gates. Let m denote the number of controlling qubits in an C^m -NOT gate (C^2 -NOT is the Toffoli gate, C^1 -NOT is the CNOT gate and C^0 -NOT is the NOT gate), then we can construct an incrementer circuit by applying the set of gates $\{C^{(n-1)}$ -NOT, $C^{(n-2)}$ -NOT, ..., C^0 -NOT $\}$ to the current state of an n -qubit system. Note that a given C^i -NOT gate, if activated, will negate qubit $i + 1$ in the circuit on which it acts. A graphical representation for $n = 6$ is shown in Figure 5.1.

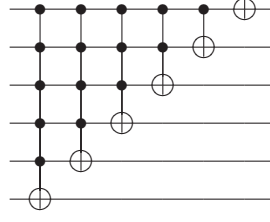


Figure 5.1: 6-qubit Binary Incrementer Quantum Circuit.

Theorem 5.1. Consider an n -qubit system with computational basis state $|b_n b_{n-1} \dots b_0 = d\rangle$, where d is the decimal value represented by binary string $b_n b_{n-1} \dots b_0$, then applying the network of gates $\text{CNOTSET}_n = \{C^{(n-1)}\text{-NOT}, C^{(n-2)}\text{-NOT}, \dots, C^0\text{-NOT}\}$ simulates Algorithm 3 and maps $|d\rangle$ to $|d + 1\rangle$.

Proof: Let CNOTSET_n denote the set of multi-controlled NOT gates $\{C^{(n-1)}\text{-NOT}, C^{(n-2)}\text{-NOT}, \dots, C^0\text{-NOT}\}$ acting on an n -qubit system. We say the CNOTSET_n is active if all the gates in CNOTSET_n are activated. Also let b_k , $0 \leq k \leq n$, denote the target qubit with state $|0\rangle$ that needs negating because all previous qubits $|b_{k-1} b_{k-2} \dots b_0\rangle$ are set to $|1\rangle$. Then CNOTSET_n will simulate the results from Algorithm 3,

- (i) since $C^{(k-1)}\text{-NOT} \in \text{CNOTSET}_n$ and $|b_{k-1} b_{k-2} \dots b_0\rangle$ are all set to $|1\rangle$, b_k is negated,
- (ii) the subset of gates $\{C^{(k-1)}\text{-NOT}, C^{(k-2)}\text{-NOT}, \dots, C^0\text{-NOT}\} \subseteq \text{CNOTSET}_n$ is active.

To see this, let $C^j\text{-NOT} \in \text{CNOTSET}_n$ be the first in the subset that is not activated.

However, by definition, the only way all previous gates $C^i\text{-NOT}$, $k - 1 \leq i \leq j + 1$, in the set could have been activated is if $C^j\text{-NOT}$ is also activated (Contradiction),

- (iii) the subset of gates $\{C^{(n-1)}\text{-NOT}, C^{(n-2)}\text{-NOT}, \dots, C^k\text{-NOT}\} \subseteq \text{CNOTSET}_n$ is not active because b_k is a common control qubit in all the gates and is set to $|0\rangle$,
- (iv) trivially, if $b_k = n$ and all qubits $|b_n, b_{n-1}, \dots, b_0\rangle$ are set to $|1\rangle$, then the entire set of gates CNOTSET_n is active and the final state of the system is $|b_n = 0, b_{n-1} = 0, \dots, b_0 = 0\rangle$.

Hence, applying CNOTSET_n to the n -qubit system produces the same results as Algorithm 3. \square

However, defining an incrementer circuit alone is not enough since we also want to simulate the movement of the particle from its current position to its adjacent vertex on the left (see Figure 3.1). Therefore, we need our QWC to also decrement the binary value of the current vertex label and map $|i\rangle$ to $|i - 1\rangle$. Given incrementer circuit A , then we can easily perform the decrement operation by applying A^\dagger (the complex conjugate of A). This is equivalent to applying the same set of gates described above but in reverse order. Note that applying circuits A and A^\dagger consecutively causes the particle to stay in the same vertex since $AA^\dagger = A^\dagger A = I$, which is equivalent to applying the identity operator.

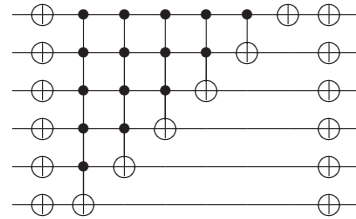
Another, more efficient approach, is to reuse the incrementer circuit to perform the decrement operation rather than constructing a separate circuit. In this case, we can decrement the binary state by incrementing its complement as shown in the simple code listed in Figure 5.2a with its corresponding circuit in Figure 5.2b. Note that, although shown for completion in Figure 5.2b, two of the three NOT gates that are applied to the least significant qubit (the top wire) are not necessary since they would cancel each other out.

Procedure QDecrement (QRegister R)

- 1: **for** each qubit in R **do**
- 2: apply NOT gate
- 3: **end for**
- 4: QIncrement(R)
- 5: **for** each qubit in R **do**
- 6: apply NOT gate
- 7: **end for**

end Procedure

(a) Decrement algorithm



(b) Decrement Circuit

Figure 5.2: Simulation of the decrementer circuit using the incrementer circuit.

To see why the decrementer circuit works, note that all we are doing by negating the incrementer circuit qubits is extracting the 2's-complement of the current state. Lets call lines 1-3 and 5-7 in Figure 5.2a operation *comp* and the current computational basis state

of the system b , then $comp(b) = -b$ and applying $QIncrement$ to $-b$ will give $-b + 1 = -1(b - 1)$. Finally, $comp(-1(b - 1)) = b - 1$, which is the result we want.

Ideally, the quantum-walk shift operator circuit should perform both of the operations described above. We can use an additional control qubit which, when set to $|1\rangle$, will activate the groups of NOT gates (as seen in Figure 5.2a) and extract the 2's-complement of the current state to perform the decrement operation. If the same control qubit is set to $|0\rangle$ then the circuit performs the increment operation as previously described since the groups of NOT gates will not be activated.

Complexity of the 1-D QWC

The 1-D QWC described previously has asymptotic linear lower bound in the number of C^m -NOT gates.

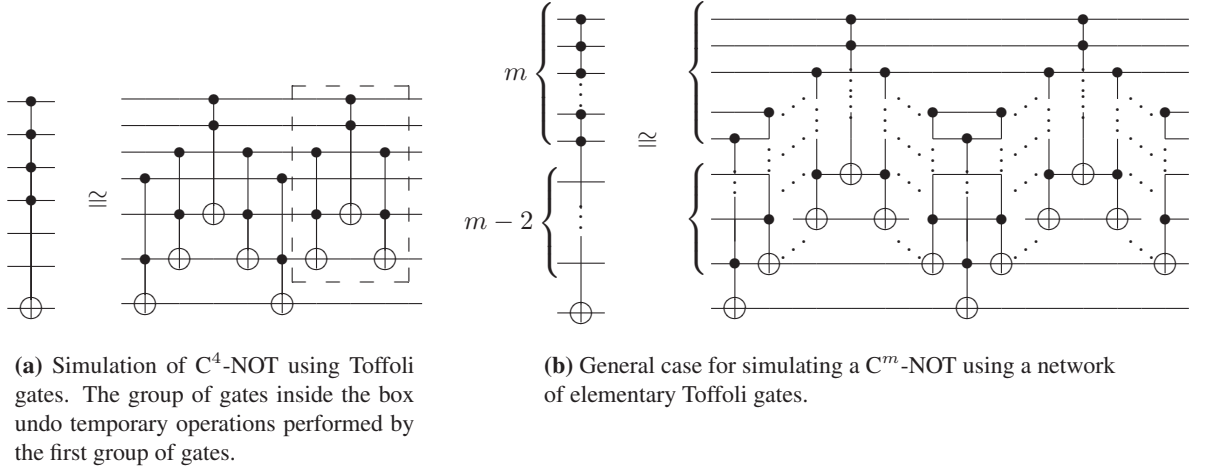
Proposition 5.2. *Consider an n -qubit system with state $|b_n b_{n-1} \dots b_0\rangle$, then applying the 1-D QWC has a gate-count lower bound of $\Omega(n)$.*

Proof: Assume an $O(1)$ cost for all quantum gates including C^m -NOT gates. Further assume that there is a circuit which uses g gates where $g < n$. Since $g < n$, then there will be at least one qubit that would not be acted on. However, performing the increment or decrement operation for all $2^n - 1$ possible binary values requires that all n qubits be flipped at least once (Contradiction). Finally, since we have to apply one C^m -NOT for each qubit $|b_j\rangle$ in the system, we get a total lower bound cost of $\sum_{j=1}^{n-1} C^{j-1}\text{-NOT} = \Omega(n)$.

□

Unfortunately, although useful for simulation purposes, C^m -NOT gates are not practical quantum constructs, meaning that they are extremely difficult to realize in a quantum-physical system. Only the elementary set of gates defined in Chapter II are considered to have $O(1)$ cost. Thus, we want to achieve improvements in implementation complexity

by synthesizing our circuit design so that only these elementary gates are used. This is achieved primarily by simulating C^m -NOT gates with the support of ancillary work qubits. Figure 5.3a shows a C^4 -NOT and its equivalent simulation using only elementary gates. Figure 5.3b shows the general C^m -NOT representation. A nice and concise proof of these equivalences is shown in [7]. We show a similar proof here for completion.



(a) Simulation of C^4 -NOT using Toffoli gates. The group of gates inside the box undo temporary operations performed by the first group of gates.

(b) General case for simulating a C^m -NOT using a network of elementary Toffoli gates.

Figure 5.3: Simulation of C^m -NOT gates using $4(m-2)$ Toffoli gates and $m-2$ work qubits. The first group of qubits identified by m are the controlling qubits while the group of qubits identified by $m-2$ are the work qubits. Note that the C^m -NOT gate is simulated correctly without pre-initialized work qubits. Also note that, after applying all the Toffoli gates, the original state of the work qubits is preserved.

First, we prove the equivalence of the network of gates shown in Figure 5.3,

Theorem 5.3. *Consider a general C^m -NOT gate that acts on a qubit system of size n , where $n \geq 5$ and $2 < m < \lceil \frac{n}{2} \rceil$. The network of Toffoli gates shown in Figure 5.3 simulates C^m -NOT.*

Proof: By inspection of the generalized gate network shown in Figure 5.3b for the C^m -NOT gate. Denote the control qubits as c_1, c_2, \dots, c_m (from top to bottom), the work qubits as w_1, w_2, \dots, w_{m-2} (from top to bottom), and the target qubit as t . w_1 is negated iff c_1, c_2 are 1, w_2 is negated iff c_1, c_2, c_3 are 1. In general, w_k is negated iff c_1, c_2, \dots, c_{k+1} are 1. If t is negated, then by definition of the Toffoli gate, w_{m-2} and c_m must be set to 1, and since

we know that w_{m-2} can only be set iff $c_1, c_2, \dots, c_{m-2+1=m-1}$ have already been set, then the entire set of control qubits must have been activated correctly. \square

Second, we prove that the number of elementary gates necessary to keep the equivalence proven in Theorem 5.3 is linear in the number of control qubits.

Theorem 5.4. *Consider a general C^m -NOT gate that acts on a qubit system of size n , where $n \geq 5$ and $2 < m < \lceil \frac{n}{2} \rceil$. The total number of Toffoli gates needed to simulate C^m -NOT is $O(m)$.*

Proof: It is trivial to see that, to maintain the structure of the network in Figure 5.3, we need to add 4 Toffoli gates and 1 work qubit for each additional control qubit that we use. This holds for $m > 2$. Thus, we need a total of $4(m - 2)$ elementary Toffoli gates to simulate C^m -NOT, which is $O(m)$. \square

Now, for a given n -qubit graph register, the largest multi-controlled gate will be C^{n-1} -NOT. To simulate C^{n-1} -NOT we need at least $(n - 1) - 2 = n - 3$ work qubits. That is quite a large number of ancilla qubits—linear in the number of circuit qubits. Fortunately, the equivalences shown in Figure 5.3 have a nice feature in that the work qubits used do not have to be preset to a specific value. Moreover, whatever the original value of the work qubits, the second set of gates (the group of gates inside the marked square depicted in Figure 5.3a) will take care of canceling the temporary operations and restoring the original value. These features allow us to reduce the number of work qubits from linear to a single constant ancilla qubit. It is easy to see from Figure 5.4 how this is accomplished. Note that, in the case of the example shown in Figure 5.4, the first C^4 -NOT gate (g_1) has enough work qubits to perform its operation. However, the work qubits used are not extra qubits, but consist of the set of controlling qubits for the second C^4 -NOT gate (g_2). We can get away with this because of the properties previously described for the network of Toffoli gates that simulate C^m -NOT gates. The result of applying g_1 is stored in the

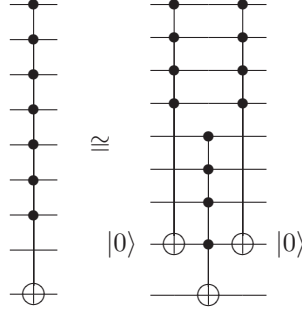


Figure 5.4: Simulation of the C^8 -NOT gate with two C^4 -NOT and one ancilla qubit. Note that the ancilla qubit must be set to $|0\rangle$ prior to the operation and will be reset to that same value upon termination of the operation.

single extra qubit that is initialized to $|0\rangle$. g_2 can now use g_1 's controlling qubits as a workspace to perform its operation. Finally, the last C^4 -NOT gate (g_3) will restore the original value of the extra qubit ($|0\rangle$) for reuse in operations later. In general, we can network smaller C^m -NOT gates to simulate the operations of larger ones while limiting the number of work qubits to a constant. Since the gate count required to simulate C^m -NOT gates using elementary gates is linear, it is trivial to see that the larger C^m -NOT gates are also simulated with linear gate count.

We have shown that simulation of C^m -NOT gates is linear in the number of control qubits m requiring $4(m - 2)$ elementary (Toffoli) gates. Using this information and Theorem 5.2 we can derive the worst-case complexity of the 1-D QWC with 1 ancilla qubit.

Theorem 5.5. *Consider a qubit system of size n with 1 ancilla qubit. We now have an $(n + 1)$ -qubit system with state $|b_{n+1}b_n \dots b_0\rangle$, then applying the 1-D QWC has upper bound $O(n^2)$ when using 1 ancilla qubits.*

Proof: Assume a cost of $4(m - 2)$ for each C^m -NOT gate, where $m > 2$. Then, in terms of m , the total cost of performing the increment/decrement operation is $2 \sum_{m=3}^{n-1} 4(m - 2) + 3$. Expanding this sum gives $2[4(n - 1 - 2) + 4(n - 2 - 2) + \dots + 4(3 - 2)] + 3$, which can easily be recognized as an arithmetic sum. Hence, the upper bound for the 2-D QWC is $O(n^2)$ \square

The option of simulating the 1-D QWC with 0 extra qubits is also available at the cost of decreasing its overall performance. Specifically, the work in [7] shows a way to simulate an arbitrary multi-controlled unitary gate with $O(n^2)$ gate-count complexity. Using this information and Theorem 5.2 we can derive the worst-case complexity of the 1-D QWC with no ancilla qubits,

Theorem 5.6. *Consider a qubit system of size n with computational basis state $|b_n b_{n-1} \dots b_0\rangle$, then applying the 1-D QWC has upper bound $O(n^3)$ when using 0 ancilla qubits.*

Proof: Consider the proven $O(n^2)$ cost for all C^m -NOT gates, where $m > 2$. Note that we consider C^2 -NOT, CNOT and NOT to be elementary gates with $O(1)$ rather than $O(n^2)$. Since, by definition, the 1-D QWC will have $\sum_{j=3}^{n-1} C^j$ -NOT gates. Then, we have a total of $\sum_{j=3}^{n-1} O(n^2) = (n-4)O(n^2) = O(n^3) \square$

Thus, we have two design options to choose from. We can trade workspace qubits for the circuit's gate count. Such trade-offs will depend on the engineering details of the quantum technology used.

5.2 Two-Dimensional QWC

The graph-space component of the QWC in two dimensions is composed of two quantum registers, one for the x dimension of the underlying lattice structure and one for the y dimension. The naive approach to designing this QWC duplicates the design of the one-dimensional design (described in Section 5.1) and applies it to the second register. This would require an additional ancilla qubit to keep the $O(n^2)$ complexity. Thus, we would have two 1-D QWCs to apply to each dimension with 2 ancilla qubits total. However, considering that we can use the qubits in the additional dimension register to work out the increment/decrement operations, we can avoid the use of ancilla qubits altogether. This improves the simulation efficiency a great deal since, in such a large multi-qubit system, a

single additional qubit will double the size of the system. However, the overall asymptotic worst-case performance of the circuit will remain the same at $O(n^2)$,

Complexity of the 2-D QWC

Theorem 5.7. *Consider an $2n$ -qubit system with 2 registers, each of size n . Label each register x and y , respectively. Then, if the system is in state $|x_n x_{n-1} \dots x_0 \otimes y_n y_{n-1} \dots y_0\rangle$, applying the 2-D QWC has upper bound $O(n^2)$.*

Proof: Assume a proven cost of $O(n^2)$ for the 1-D QWC. Since we use exactly 1 1-D QWC for each of the x and y registers, we have a total upper bound cost of $2 * O(n^2) = O(n^2)$ \square

Each of the designs for the QWCs discussed in this chapter have been implemented in `QuWalkLib`. The quantum walk on a line/cycle described in Section 4.5 was implemented using the design for the 1-D QWC described in Section 5.1. The quantum walk on a 2-D lattice was also implemented in the `QuWalkLib` `QWalk` class using the design described in Section 5.2.

5.3 Speed-ups of the Quantum Walk

The asymptotic complexity of the classical 2-D random walk algorithm, when used to solve particular problems (2-SAT, Triangle Finding in a graph, Element Distinctiveness, etc.), is usually $O(N^2)$ [23], where N is the number of vertices. In the quantum case, there is an exponential improvement in the number of resources that are required to represent N number of vertices. This was shown in Section 3.1, where the number of vertices was defined as 2^g , where g is equal to the number of qubits in the system. Hence, the quantum case has complexity $O(\log^2(N))$ in the number of vertices. Furthermore, the walk propagates quadratically faster, as shown in [5]. However, there is a setback in the quantum case related to measurement and extraction of solutions for particular applications of quantum

walks to solve a wider range of problems. Specifically, a lot of classical algorithms involving random walks rely on the Gaussian distribution of vertices to extract the correct solution to a problem. However, as seen in Section 4.5, the probability distribution of the quantum walk is strikingly different from the Gaussian distribution. This makes it difficult to extract solutions in the same fashion as in the classical case, and requires new techniques that accommodate the particular properties of quantum walks. The next and final chapter will show an example of the problem we just described along with several proposals on how to proceed with future work in the area of quantum walks and their algorithmic applications.

CHAPTER VI

Future Work: Simulation-based Algorithm Design with Quantum Walks

Although quantum walks have been mathematically proven to be more efficient than classical random walks, not many of the useful applications of classical random walks have translated to the quantum case. This is not unique to quantum walks, and is in fact true of most classical algorithms that researchers have redesigned in the quantum context. This is part of the reason for the relative lull in the development of new quantum algorithms [29]. Furthermore, this is also the reason why there are a lot of open problems in the field of Quantum Computation and why most problems are stated in an abstract way without much details of physical designs.

In this vein, the subject of this chapter is to present an overview of how quantum walks may be used to solve a particular problem. Specifically, we assess the potential of quantum walks for solving the *Heat Equation*. We discuss a preliminary formulation of the problem and determine abstract ideas for addressing implementation issues. Finally, we close our discussion with a look at future work in the area of simulation of quantum walks.

6.1 Quantum-Walk Solution to the Heat Equation

Classical random walks can be used to solve systems of linear equations. Although, in general, random walks are quite inefficient at this task, under special conditions, they

can actually perform better than other algorithms. Specifically, previous work [23] has shown that reduction of a PDE to a set of linear equations can produce such a system. We now consider a sample problem to demonstrate a possible formulation of a quantum-walk algorithm to find the numerical solution of a PDE. We compare the quantum formulation against the classical one and identify implementation issues.

6.1.1 Definition of the Heat-Equation Problem

Consider a 10cm (width) by 30cm (height) plate. Assume that three of the plate's edges are heated to a constant 0°F temperature, and that only the top edge is heated to a constant 100°F temperature. We want to find the steady-state temperature distribution T of the plate. The continuous solution follows Laplace's Equation in 2-dimensions,

$$(6.1) \quad \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

where $0 \leq x \leq N = 10$ is the width and $0 \leq y \leq M = 30$ is the height of the plate. The given temperatures at each of the edges of the plate describe the 4 boundary conditions,

$$T_{0,y} = 0 \quad T_{x,0} = 0 \quad T_{10,y} = 0 \quad T_{x,30} = 100$$

The standard finite-difference approximation to each of the second-order derivatives in Equation 6.1 are,

$$(6.2) \quad \frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{(\Delta x)^2}$$

$$(6.3) \quad \frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{(\Delta y)^2}$$

where i and j are indices moving along the x (width) and y (height) dimensions, respectively. Now we use a square grid to describe the positions on the plate such that $\Delta x = \Delta y = h$. Adding the right-hand sides of Equations 6.2 and 6.3 will give,

$$(6.4) \quad \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j}}{h^2} = 0$$

Equation 6.4 allows us to arrange the difference equations of each unknown value $T_{i,j}$ into a linear system of the form $Au = b$ where u is the vector containing $(M - 1)(N - 1)$ unknowns, b is the vector of boundary values of T , and A is a large sparse matrix containing up to $[(N - 1)(M - 1)]^2$ elements.

As an example of the construction of this matrix equation, let's reduce the size of our grid to 4×4 , but retain all other properties of the problem. The finite-difference equations for the interior points of the new grid are,

$$T_{3,2} + T_{1,2} + T_{2,3} + T_{2,1} + 4T_{2,2} = 0$$

$$T_{3,2} + T_{1,2} + T_{2,3} + T_{2,1} + 4T_{2,2} = 0$$

$$T_{3,2} + T_{1,2} + T_{2,3} + T_{2,1} + 4T_{2,2} = 0$$

$$T_{3,2} + T_{1,2} + T_{2,3} + T_{2,1} + 4T_{2,2} = 0$$

However, note that 8 of the points above are known boundary values. This allows us to build the following matrix equation,

$$\begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix} \begin{bmatrix} T_{2,2} \\ T_{3,2} \\ T_{2,3} \\ T_{3,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -100 \\ -100 \end{bmatrix}$$

which can be solved for $u = A^{-1}b$ using several known algorithms.

6.1.2 Classical Solution using Random Walks

For a small number of grid points, the matrix equation approach described above may be used to solve the problem directly by Gaussian Elimination, Gauss-Jordan Elimination or Cholesky Factorization. However, for relatively modest N and M , the storage and computational costs of these algorithms become prohibitive. So, in practice, more efficient methods are used to approach the linear system from Equation 6.4. *Iterative* methods such as Conjugate Gradient, Jacobi Iteration, Gauss-Seidel Iteration or Successive Over-Relaxation are often the algorithms of choice. To understand how iterative methods work we rearrange Equation 6.4 and solve for $T_{i,j}$,

$$(6.5) \quad T_{i,j} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4}$$

In general, iterative methods work by setting all boundary grid points to their known values, and the interior points (unknowns) to some arbitrary approximate value. We then apply Equation 6.5 repeatedly, finding approximate solutions to each of the unknown values on each iteration. The algorithm stops when the unknown values start converging to a constant.

Note that Equation 6.5 makes the value of T at a given point (i, j) dependent on the values of its adjacent points (up, down, left, right). Specifically, each $T_{i,j}$ is going to be an average (expected) value of its surrounding points. This property suggests that the solution to each $T_{i,j}$ can also be approximated by random-walk methods. The proof of this assertion is beyond the scope of this paper and we refer the reader to [23]. In particular, we can formulate a random walk directly from Equation 6.5 such that the walker behaves in a manner equivalent to the equations in the linear system. Formally, we define a 2-dimensional array of size $(N - 1)(M - 1)$ that represents the points (vertices) on the plate

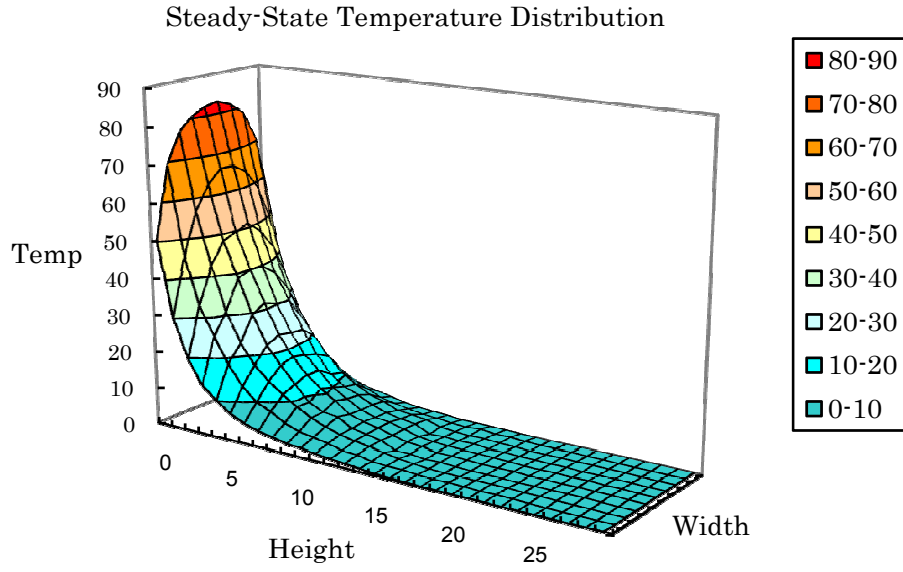


Figure 6.1: Steady-state temperature distribution for 10 x 30 plate with 100° heat applied on the top edge and 0° applied on the remaining three edges.

for which we want to approximate a solution. The random walker will then move from vertex (i, j) to any of its 4 adjacent vertices with probability $1/4$.

Once the random walk has been formulated, the approximate solution for a vertex $T_{i,j}$ is found by starting the walk at that point. The walker will then propagate linearly with time until it reaches a boundary. Once the walker reaches a boundary it is absorbed, and the boundary value is accumulated. Note that absorption of the walker will always happen in the classical case [14]. By the Law of Averages, performing several walks and averaging over the encountered boundary values will give us an approximate solution to within a chosen accuracy ε . Appendix D shows an implementation of this random walk in C++ and Figure 6.1 shows the graphical representation of the output from the program.

6.1.3 Quantum-Walk Formulation of the Heat-Equation Problem

Having identified classical random walks as a method for solving linear systems, we may ask, can a quantum walk formulation perform better? This is certainly a question that we hope to answer as part of our research work. As a first step towards answering this question, we discuss a preliminary quantum formulation of the classical random walk defined above.

The quantum case requires considerably more work to formulate, compared to the classical case. We implement a discrete quantum walk for Equation 6.5 using a square lattice structure on an n -qubit graph space where $n = \log((N - 1)(M - 1))$. Since a square lattice is 4-regular, the coin toss operation can be implemented using a degree-4 DFT coin,

$$(6.6) \quad C_{DFT}^4 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

which, when applied to a 2-qubit coin space, produces a symmetrical and unbiased superposition of all four vertices adjacent to (i, j) . Note that, since we are using a symmetrical coin operator, we have no need to initialize the coin space prior to performing the walk. Furthermore, we define the shift operator to make the following mappings,

$$(6.7) \quad \begin{aligned} S : \quad |\rightarrow\rangle \otimes |i, j\rangle &\longrightarrow |\rightarrow\rangle \otimes |i + 1, j\rangle \\ |\leftarrow\rangle \otimes |i, j\rangle &\longrightarrow |\leftarrow\rangle \otimes |i - 1, j\rangle \\ |\uparrow\rangle \otimes |i, j\rangle &\longrightarrow |\uparrow\rangle \otimes |i, j + 1\rangle \\ |\downarrow\rangle \otimes |i, j\rangle &\longrightarrow |\downarrow\rangle \otimes |i, j - 1\rangle \end{aligned}$$

This formulation is not unique and, as shown in [30], a different choice of coin operator (*Hadamard* and *Grover* coins could also be used) and shift operator could lead to different

results. In our formulation, given an initial state $|T_{0,0}\rangle \otimes |\uparrow, \rightarrow\rangle$, the first iteration of the walk would look like,

$$\begin{aligned}
 |T_{0,0}\rangle \otimes |\uparrow, \rightarrow\rangle &\xrightarrow{C_{DFT}^4} \frac{1}{\sqrt{4}}(|\downarrow, \leftarrow\rangle - |\uparrow, \rightarrow\rangle + i|\downarrow, \rightarrow\rangle + i|\uparrow, \leftarrow\rangle) \otimes |T_{0,0}\rangle \\
 (6.8) \quad &\xrightarrow{S} \frac{1}{2}|\downarrow, \leftarrow\rangle \otimes |T_{-1,-1}\rangle - \frac{1}{2}|\uparrow, \rightarrow\rangle \otimes |T_{1,1}\rangle \\
 &\quad + \frac{i}{2}|\downarrow, \rightarrow\rangle \otimes |T_{1,-1}\rangle + \frac{i}{2}|\uparrow, \leftarrow\rangle \otimes |T_{-1,1}\rangle
 \end{aligned}$$

6.1.4 Measurement and Extraction of the Solution

Note that in the quantum formulation, as shown in Equation 6.8, we are able to construct a superposition of the states of the walk. This makes the spread of the walk quadratically faster in time [18], as compared to the classical case, which points to a faster-than-classical solution. However, the role of an absorbing boundary in the quantum case is not quite clear. It has been shown in [22] and [6] that a quantum walker is not absorbed 100 percent of the time. Moreover, absorbing a walker at boundary b requires partial measurement of the system at state $|b\rangle$. This means that after measurement M_b we will find the system in the state $|b\rangle$ with probability p_b (see Section 3.2). This implies that we might actually need to perform more walks than the classical case to be able to obtain an good average (since we won't get the boundary value all the time the walker is absorbed), and get a numerical estimate of the solution. A possible way to resolve this issue is to somehow calculate the average without destroying the superposition, and then store the number on a separate qubit before performing the measurement.

Another issue related to the absorption probability of the walker is the probability distribution of vertices in a quantum walk. This issue was briefly touched upon in Section 4.5. This is a fundamental obstacle in extracting the correct solution of the heat equation

from the quantum walk. One possible way of overcoming this obstacle is based on the work in [20] and [19]. Interestingly, the authors show that introducing a small amount of decoherence into a quantum walk will change the shape of the probability distribution of the vertices. Even more surprising is the fact that the speed-up of the quantum walk is not lost. Hence, one can potentially introduce a precise amount of decoherence so that the probability distribution looks closer to a uniform distribution. If the vertices are distributed uniformly at the “quantum level”, we can think of several numerical-analysis techniques, such as Monte Carlo methods, to calculate an approximate solution to the problem using an iterative projective-measurement-based quantum algorithm. However, whether such an algorithm would yield overall performance improvements over classical Monte Carlo methods is still an open question. We intend to continue research work in this area in the future.

6.2 Improving High-Level Quantum Programming

As part of our continuing work in the area of quantum walks, we intend to make improvements to `QuWalkLib`. Specifically, we want to include additional functionality that enables the following:

Pre-compiled quantum gate and circuit library. Primarily, we want to extend the number of pre-compiled quantum gates in the `QuWalkLib` library. It is now currently limited to the NOT, CNOT, Toffoli and Hadamard gates, which are the gates that are necessary to implement quantum walks. However, there are other important quantum gates that are useful when building high-level quantum programs. We want to give the programmer access to these gates so that better circuits can be designed using `QuWalkLib`. Also, a pre-compiled set of quantum circuits, such as the n -qubit incrementer/decrementer, 3-qubit majority-and-add, n -qubit binary-to-gray-

code converter, would be equally useful.

Circuit optimization techniques. Chapter V demonstrated how important it is for a high-level quantum programmer to pay attention to circuit design when developing quantum algorithms. Although, the circuit synthesis work that was described in Chapter V was done in an add-hoc basis, in practice, we want to use proven logic synthesis algorithms that can greatly improve our circuit design. The authors of [24] describe a particular quantum logic synthesis algorithm that can be readily implemented in `QuWalkLib`. We hope to incorporate this functionality into the `QuWalkLib` `QPCircuit` class.

Additional quantum-walk graph structures. The `QWalk` class is currently limited to three common graphs – line, cycle, and grid. However, considerable work in quantum walks has been devoted to hypercube structures and general graphs. Future versions of `QuWalkLib` will include implementations that allow a high-level quantum programmer to experiment with simulations on all these structures.

Metrics for evaluating quantum walks. Finally, additional functions that calculate real-time metrics inside a quantum walk is an ideal feature to have. This helps a high-level quantum programmer calculate specific numerical data that can improve his or her understanding of the dynamics of a quantum walk. This feature will be key in continuing the preliminary research described in Section 6.1.

APPENDICES

APPENDIX A

QuIDDP Pro Script – Quantum Walk on a Square

Note: QuIDDP Pro comments are defined by the # character

```

1  # Initialize variables
2  n_qbits = 3;
3  itrs = 8;
4  q0 = cb("0");
5  q1 = cb("1");
6  state_vector = 1;
7  coin_flip = 1;
8  shift_p1 = 1;
9  shift_p2 = 1;
10
11 # Create the state vector of the system
12 j = 1;
13 while (j <= n_qbits)
14     state_vector = kron(state_vector, q0);
15     j = j + 1;
16 end
17
18 # Create the coin operator
19 coin_flip = kron(hadamard(1), identity(2));
20
21 # Create shift operators
22 shift_p1 = kron(cnot("cx"), identity(1));
23 shift_p2 = cu_gate(sigma_x(1), "n1x3", 3);
24
25 # Do the walk
26 j = 1;

```

```
27 while(j <= itrs)
28     # Perform coin flip
29     state_vector = coin_flip * state_vector;
30
31     # Perform shift position
32     state_vector = shift_p1 * state_vector;
33     state_vector = shift_p2 * state_vector;
34
35     j = j + 1;
36 end
37
38 final_state_vector = state_vector
```

APPENDIX B

QuIDDP_{ro} Script – Quantum Walk on a Square with Decoherence

Adding these statements to the script in Appendix A between lines 33 and 35 will simulate decoherence and produce a quantum walk with classical random walk behavior.

```
34      # Measure the position qubits  
35      state_vector = measure_sv(2, state_vector);  
36      state_vector = measure_sv(3, state_vector);
```


APPENDIX C

QuWalkLib Program – Quantum Walk on a Cycle

```

1  // Requires: iostream, map, QPWalk
2  // Description:
3  // Simulation of the quantum walk on a line
4
5  #include <iostream>
6  #include <map>
7  #include "QPWalk.h"
8
9  using namespace Quiddpro;
10 using namespace std;
11
12 int main(int argc, char *argv[])      {
13     // create the quantum walk object
14     // parameters:
15     //     false = 1-dim walk (true = 2-dim walk)
16     //     32 = total number of vertices
17     //     "mywalk" = quiddpro variable name
18     QPWalk awalk(false, 32, "mywalk");
19
20     // initialize the walk to an unbiased superposition
21     awalk.init();
22
23     // run the quantum walk for 100 iterations
24     awalk.run(100);
25
26     // get the probability distribution for the
27     // x-dimension register and print to cout

```

```
28     vector<double> pdist = awalk.get_xdim_prob_dist();
29     vector<double>::iterator ditr = pdist.begin();
30     while(ditr != pdist.end()) {
31         cout << *ditr << "_";
32         ditr++;
33     }
34     return 0;
35 }
```

APPENDIX D

Random walk implementation in C++

Random walk implementation to approximate the solution of the problem defined in Section V.

```
1  #include <stdio.h>
2  #include <tchar.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <assert.h>
6  #include <stdio.h>
7  #include <time.h>
8  #include <iostream>
9  #include <fstream>
10 #include <iomanip>
11
12 using namespace std;
13
14 class CustomRandom {
15     public:
16
17         CustomRandom() {
18             srand( 0 );
19         }
20
21         CustomRandom(int seed) {
22             srand( seed );
23         }
24
```

```

25     double Random() {
26         return ( (double) rand() / (double) RAND_MAX );
27     }
28 };
29
30 double rand_walk(int height, int width,
31                 int samplesize, int x, int y, double **T)
32 {
33     CustomRandom randgen( time(0) );
34
35     // current position of the walk
36     int curr_x, curr_y;
37
38     // flag to check whether walker has been absorbed
39     bool absorbed;
40
41     double coin;
42
43     // do the walk
44     for( int k = 0; k < samplesize; k++ )    {
45
46         // start position of the random walk
47         curr_x = x;
48         curr_y = y;
49
50         absorbed = false;
51
52         // keeping walking until absorbed
53         while( !absorbed )    {
54             coin = randgen.Random();
55             if( coin < 0.25 )
56                 curr_x += 1;
57             else if( coin < 0.5 && coin >= 0.25 )
58                 curr_y +=1;
59             else if( coin < 0.75 && coin >= 0.5 )
60                 curr_x -= 1;
61             else

```

```

62         curr_y -= 1;
63
64         if( curr_x == 0 || curr_x == height - 1 ||
65             curr_y == 0 || curr_y == width - 1 )
66         {
67             T[x][y] += T[curr_x][curr_y];
68             absorbed = true;
69         }
70     }
71     // return average of runs
72     return T[x][y] = T[x][y] / samplesize;;
73 }
74
75 int main( int argc, char **argv )    {
76
77     // Constants
78     const double PI = 3.1415;
79     const int samplesize = 1000;
80     const int width = 10 + 2; // add two for boundaries
81     const int height = 30 + 2; // add two for boundaries
82
83     // temperature function
84     double **T;
85
86     T = new double *[height];
87     for(int i = 0; i < height; i++ )
88         T[i] = new double [width];
89
90     // set up boundary conditions
91     for( int i = 0; i < height; i++ )    {
92         T[i][0] = 0;
93         for(int j = 1; j < width - 1; j++)
94             T[i][j] = 0.0;
95         T[i][width - 1] = 0;
96     }
97     for( int j = 0; j < width; j++ )    {
98         T[0][j] = 100;

```

```

99         T[height - 1][j] = 0;
100     }
101
102     // perform 1000 random walks for each point on the grid
103     for( int i = 1; i < height - 1; i++ )    {
104         for( int j = 1; j < width - 1; j++ )    {
105             T[i][j] = rand_walk(height, width,
106                                 samplesize, i, j, T);
107         }
108     }
109
110     // output solution to a file
111     ofstream ofstr;
112     ofstr.open( "solution.dat", ios::out );
113     ofstr << "\t";
114     for( int i = 1; i < width - 1; i++ )    {
115         ofstr << i << "\t";
116     }
117     ofstr << endl;
118     for( int i = 1; i < height - 1; i++ )    {
119         ofstr << i << '\t';
120         for( int j = 1; j < width - 1; j++ )    {
121             ofstr << T[i][j] << "\t";
122         }
123         ofstr << endl;
124     }
125     ofstr.close();
126
127     //clean up
128     for( int i = 0; i < height; i++ )
129         delete [] T[i];
130     delete [] T;
131
132     return 0;
133 }

```

BIBLIOGRAPHY

BIBLIOGRAPHY

- [1] N. A. and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
- [2] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. arXiv.org: quant-ph/0012090, 2001.
- [3] Y. Aharonov, L. Davidovich, and N. Zagury. *Physical Review*, 48(1687), 1993.
- [4] A. Ambainis. Quantum walk algorithm for element distinctness. arXiv.org: quant-ph/0311001, 2004.
- [5] A. Ambainis, J. Kempe, and A. Rivosh. Coins make quantum walks faster. arXiv.org: quant-ph/0402107, 2004.
- [6] E. Bach, S. Coppersmith, M. Goldschen, R. Joynt, and J. Watrous. One-dimensional quantum walks with absorbing boundaries. arXiv.org: quant-ph/0207008, 2002.
- [7] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. arXiv.org: quant-ph/9503016, 1995.
- [8] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by quantum walk. arXiv.org: quant-ph/0209131, 2002.
- [9] D. P. DiVincenzo. Two-bit gates are universal for quantum computation. arXiv.org: cond-mat/9407022, 1995.
- [10] A. Flitney, D. Abbott, and N. Johnson. Quantum random walks with history dependence. arXiv.org: quant-ph/0311009, 2004.
- [11] A. Flitney, D. Abbott, and J. Ng. Quantum parrondo's games. arXiv.org: quant-ph/0201037, 2002.
- [12] J. Grabbe. An introduction to quantum game theory. arXiv.org: quant-ph/0506219, 2005.
- [13] L. Grover. Quantum mechanics helps in searching for a needle in a haystack. arXiv.org: quant-ph/9706033, 1997.
- [14] B. I. Henry and M. T. Batchelor. Random walks on finite lattice tubes. arXiv.org: math-ph/0305023, 2003.
- [15] <http://vlsicad.eecs.umich.edu/Quantum/>.
- [16] <http://www.eecs.umich.edu/hjgarcia/>.
- [17] J. Kempe. Quantum random walks hit exponentially faster. arXiv.org: quant-ph/0205083, 2002.
- [18] J. Kempe. Quantum random walks: an introductory overview. arXiv.org: quant-ph/0303081, 2003.
- [19] V. Kendon and B. Tregenna. Decoherence is useful in quantum walks. arXiv.org: quant-ph/0209005, 2002.

- [20] V. Kendon and B. Tregenna. Decoherence in discrete quantum walks. arXiv.org: quant-ph/0301182, 2003.
- [21] D. Meyer. From quantum cellular automata to quantum lattice gases. arXiv.org: quant-ph/9604003, 1996.
- [22] A. Nayak and A. Vishwanath. Quantum walk on the line. arXiv.org: quant-ph/0010117, 2000.
- [23] K. Sabelfeld and N. Simonov. *Random Walks on Boundary for solving PDEs*. Brill Academic, 1994.
- [24] V. Shende, S. Bullock, and I. L. Markov. Synthesis of quantum logic circuits. arXiv.org: quant-ph/0406176, 2006.
- [25] V. V. Shende, A. K. Prasad, K. N. Patel, I. L. Markov, and J. P. Hayes. Reversible logic circuit synthesis. arXiv.org: quant-ph/0207001v4, 2003.
- [26] N. Shenvi. Random walk simulations.
- [27] N. Shenvi, J. Kempe, and K. Whaley. Quantum random-walk search algorithm. arXiv.org: quant-ph/0210064, 2003.
- [28] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. arXiv.org: quant-ph/9508027, 1997.
- [29] P. Shor. Progress in quantum algorithms. 2005.
- [30] B. Tregenna, W. Flanagan, R. Maile, and V. Kendon. Controlling discrete quantum walks: coins and initial states. arXiv.org: quant-ph/0304204, 2003.
- [31] G. F. Viamontes, I. L. Markov, and J. P. Hayes. Improving gate-level simulation of quantum circuits. arXiv.org: quant-ph/0309060, 2003.
- [32] J. Watrous. Quantum simulations of classical random walks and undirected graph connectivity. arXiv.org: cs.CC/9812012, 2001.
- [33] T. Yamasaki, H. Kobayashi, and H. Imai. Analysis of absorbing times of quantum walks. arXiv.org: quant-ph/0205045, 2003.