Efficient Quantum Circuit Simulation

by

George F. Viamontes

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2007

Doctoral Committee:

Professor John P. Hayes, Co-Chair
Associate Professor Igor L. Markov, Co-Chair
Professor Christopher R. Monroe
Associate Professor Scott A. Mahlke
Assistant Professor Yaoyun Shi

To my family and friends

ACKNOWLEDGEMENTS

I would like to thank many people who were instrumental in helping me finish my Ph.D. My advisors John Hayes and Igor Markov provided me with excellent ideas to pursue, offered endless advice and comments on every publication we worked on together, and pushed me to do a great deal of solid research over the years.

Maintaining one's sanity is also a key part of surviving graduate school, and my friends at Michigan played a major role in that regard. In no particular order, I deeply thank David Papa, Jarrod Roy, Aaron Ng, Arathi Ramani, DoRon Motter, Saurabh Adya, Julia Lipman, Steve Plaza, Smita Krishnaswamy, Jin Hu, Manoj Rajagopalan, Kai-hui Chang, James Lu, Colleen Craig, my friends from Los Alamos National Laboratory, and last but certainly not least, Patrick Shea.

I never would have gone to graduate school had I not been influenced earlier on in life to pursue engineering and an advanced degree. My parents played a big role in this regard, and I still remember my dad asking me to come up with other uses for a fork aside from eating when I was a child. Also, many of my good friends from my undergraduate days at Notre Dame went to graduate school in various engineering disciplines, which helped make going to graduate school seem like less of an alien concept. Lastly, I would like to thank my entire family and my other close friends from St. Louis, particularly Matt and Derek, for all the support over the years.

Many brain cells died in the making of this Ph.D. Their noble sacrifice will not be forgotten.

PREFACE

Quantum-mechanical phenomena are playing an increasing role in information processing as transistor sizes approach the nanometer level, while the securest forms of communication rely on quantum data encoding. When they involve a finite number of basis states, these phenomena can be modeled as quantum circuits, the quantum analogue of conventional or "classical" logic circuits. Simulation of quantum circuits can therefore be used as a tool to evaluate issues in the design of quantum information processors. Unfortunately, simulating such phenomena efficiently is exceedingly difficult. The matrices representing quantum operators (gates) and vectors modeling quantum states grow exponentially with the number of quantum bits.

The information represented by quantum states and operators often exhibits structure that can be exploited when simulating certain classes of quantum circuits. We study the development of simulation methods that run on classical computers and take advantage of such repetitions and redundancies. In particular, we define a new data structure for simulating quantum circuits called the quantum information decision diagram (QuIDD). A QuIDD is a compressed graph representation of a vector or matrix and permits computations to be performed directly on the compressed data. We develop a comprehensive set of algorithms for operating on QuIDDs in both the state-vector and density-matrix formats, and evaluate their complexity. These algorithms have been implemented in a general-purpose simulator program for quantum-mechanical applications called QuIDDPro. Through extensive experiments conducted on representative quantum simulation

applications, including Grover's search algorithm, error characterization, and reversible circuits, we demonstrate that QuIDDPro is faster than other existing quantum-mechanical simulators such as the National Institute of Standards and Technology's QCSim program, and is far more memory-efficient. Using QuIDDPro, we explore the advantages of quantum computation over classical computation, simulate quantum errors and error correction, and study the impact of numerical precision on the fidelity of simulations. We also develop several novel algorithms for testing quantum circuit equivalence and compare them empirically. The QuIDDPro software is equipped with a user-friendly interface and is distributed with numerous example scripts. It has been used as a laboratory supplement for quantum computing courses at several universities.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

CHAPTER I

Introduction

Richard Feynman observed in the 1980s that simulating quantum mechanical processes on a standard *classical* computer seems to require super-polynomial memory and time [35]. For instance, a complex vector of size 2^n is needed to represent all the information in the quantum version of an n-bit vector denoting a quantum mechanical state, and square matrices of size 2^{2n} are needed to model (simulate) the time evolution of states [51]. Consequently, Feynman proposed quantum computing, which uses the quantum mechanical states themselves to simulate quantum processes. The key idea is to replace bits with quantum bits called qubits as the fundamental units of information. A quantum computer can operate directly on exponentially more data than a classical computer with a similar number of operations and information units. Thus in addressing the problem of simulating quantum mechanical processes more efficiently, Feynman discovered a new computing model that, as was subsequently shown, can outperform the best known classical computational methods for certain problems.

Since Feynman's seminal work, a number of practical information processing applications that exploit quantum mechanical effects have been proposed. Quantum compu-

tational algorithms have been discovered to quickly search unstructured databases [33] and to factor numbers in polynomial time [65]. Implementing quantum algorithms has proved to be particularly difficult, however, in part due to errors caused by the environment [41, 50]. Additionally, quantum mechanics has been harnessed for secure key exchange in encrypted communication since the act of eavesdropping can be detected as destructive measurement on quantum states [6, 7, 27]. Another related application is the design of reversible logic circuits. The logic operations performed on qubits in quantum computation must be unitary, so they are all invertible and allow re-derivation of the inputs given the outputs [51]. This phenomenon gives rise to a host of potential applications in fault-tolerant computation. Since reversible logic, secure quantum communication, and quantum algorithms can be modeled as quantum circuits [51], the quantum analogue of digital logic circuits, quantum circuit simulation could be of major benefit to these applications. In fact, any quantum mechanical phenomenon with a finite number of states can be modeled as a quantum circuit [51, 13]. Unfortunately, the very problem which brought forth quantum mechanics as a useful computational tool is the same problem which, in general, renders quantum circuit simulation on a classical computer intractable.

Software simulation has long been an invaluable tool for the design and testing of classical, digital circuits. This problem, typically considered as a computer-aided design (CAD) task, was once thought to be computationally intractable as well. Early simulation and synthesis techniques for n-bit circuits often required $O(2^n)$ runtime and memory, with the worst-case complexity being fairly typical. Later algorithmic advancements brought about the ability to perform circuit simulation much more efficiently in practical cases.

One such advance was the development of a data structure called the reduced ordered binary decision diagram (ROBDD) [17], which can greatly compress the Boolean description of digital circuits and allow direct manipulation of the compressed form. Software simulation may also play a vital role in the development of quantum hardware by enabling the modeling and analysis of large-scale designs that cannot be implemented physically with current technology. Unfortunately, straightforward simulation of quantum designs by classical computers executing standard linear-algebraic routines requires $O(2^n)$ time and memory [35, 51]. However, just as ROBDDs and other innovations have made the simulation of very large classical computers tractable, new algorithmic techniques can allow the efficient simulation of quantum computers in many important cases.

1.1 Goals of Quantum Circuit Simulation

Interestingly, if a classical computer can simulate a quantum computer solving a particular problem, then this implies that a classical computer is computationally as powerful as a quantum computer for the problem in question. Therefore, by discovering new classical algorithms which can efficiently simulate quantum computers in certain cases, we are probing the limitations of quantum computing. In light of this, it might seem that simulation for the sake of improving quantum hardware introduces competing goals. However, we argue that error characterization and error correction schemes developed with the aid of efficient classical simulation can in principle be applied to other quantum circuits which cannot be simulated efficiently. In addition, the automatic creation and optimization of quantum circuits for various tasks, also known as quantum circuit synthesis, can make use of classical simulation.

In this work we describe the development of practical software methods which enable such simulation and propose extensions to these methods to encompass an even larger set of simulatable quantum circuits. Such simulation will be used as a tool to address the following issues:

1. Characterizing the effect of various errors in practical quantum circuits.
2. Testing multi-qubit error correction techniques to cope with such errors.
3. Verifying the correctness of synthesized quantum circuits.
4. Exploring the boundaries between the quantum and classical computational models.

We have completed a large body of work addressing these topics. This work includes the development of the quantum information decision diagram (QuIDD), which facilitates efficient simulation and equivalence checking of a non-trivial class of quantum circuits [76, 78, 79, 80, 82, 83, 81, 77]. However, before delving into a survey of simulation techniques in Chapter II, it is instructive to first review some background information on quantum computation and a few classical CAD data structures. All simulation techniques described in this dissertation are exact up to machine precision and use no approximations.

1.2 Background

Without assuming prior knowledge of quantum computing, the first two subsections outline the basic concepts required to understand this work. The third subsection provides background on the reduced ordered binary decision diagram data structure, which is required to understand our preliminary work involving the quantum information decision diagram.

1.2.1 Quantum Mechanics

The physics underlying quantum computing is quantum mechanics. To grasp the basics of quantum computing, a brief overview of the important properties of quantum mechanics is in order. Although there is more than one model of quantum mechanics, we choose to restrict the presentation to the *Dirac* model which makes extensive use of linear algebra.

The Fundamental Postulates

Quantum mechanics, and therefore quantum computing, is governed by four fundamental postulates that have been verified over the years through a number of experiments [59]. Any simulation of quantum computing must implement these four postulates in some form if true quantum behavior is to be modeled. A brief summary of the four postulates follows (these postulates can be found in a number of standard quantum mechanical texts including [51, 59]).

Postulate 1. Quantum states are represented as vectors in a Hilbert space.

Since the vectors that arise in quantum computing have finite sizes, the Hilbert space of quantum states is simply a complex-valued vector space for which the *inner product* is defined. To recall from linear algebra, the inner product of two vectors x and y is

$$\Sigma_{i=1}^{n} x_i^* y_i = [x_1^* \ldots x_n^*] \begin{bmatrix} z_1 \\ . \\ . \\ . \\ z_n \end{bmatrix}, \tag{1.1}$$

where a^* denotes the complex conjugate of a complex number a.

For our purposes, this means that qubits, which are quantum states, are represented as vectors for which we can compute inner products. The need for a vector representation comes from a physical phenomenon called superposition. In quantum computing, two low energy stable states are used to represent the classical values 0 and 1 and are referred to as *computational basis states* [51]. Like an analog signal, the range of qubit values is an infinite continuum of values between 0 and 1. However, unlike an analog signal, these values denote a *probability* of obtaining a 0 or 1 upon measurement of a qubit. This is the essence of superposition. More formally, given a state vector for some qubit $|\psi\rangle$ [1] $= \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$, α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$. α and β are *probability amplitudes*, such that $|\alpha|^2$ and $|\beta|^2$ are the probabilities of measuring the qubit as a 0 and as a 1, respectively. One can think of α as the amount of "zeroness" and β as the amount of "oneness" that the qubit contains. In an equal superposition, $|\alpha|^2 = |\beta|^2$, and a qubit in such a state is interpreted as being both 0 and 1 simultaneously. Mathematically, an equal superposition of one qubit has the form $|+\rangle = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$. It is easy to see that $|\frac{1}{\sqrt{2}}|^2 + |\frac{1}{\sqrt{2}}|^2 = 1$. Similarly, the basis states 0 and 1 have the form $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

As will be demonstrated shortly, the massive parallelism achieved in quantum computing is due largely to both the property of superposition and Postulate 4. Furthermore, since the state vectors associated with qubits exist in a finite-dimensional Hilbert space, their inner products must be defined as per the definition of a finite-dimensional Hilbert space.

[1] $|x\rangle$ denotes a *ket* in the standard Dirac notation and is short-hand for a complex-valued column vector representing a quantum state.

This property is shown to be important in both Postulate 3 and the no-cloning theorem, which are discussed later.

Postulate 2. Operations on quantum states in a closed system are represented using matrix-vector multiplication of the quantum state vector by a unitary matrix.

This postulate describes special types of matrices that are analogous to logic gates in classical computation. A unitary matrix has the property that its adjoint equals its inverse. The adjoint of a matrix is simply the complex conjugate transpose. In other words, given a matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, its adjoint is $\begin{bmatrix} a^* & c^* \\ b^* & d^* \end{bmatrix}$.

Unitary matrices are *operators* which can be used to modify the values of qubits like logic gates. For the remainder of this proposal, the terms operator and gate are used interchangeably. Unlike classical logic gates, however, all quantum operators are reversible. Given a sequence of operations performed on a set of qubits, the qubits can be returned to their original state simply by performing the inverse of each operation in the reverse order that the operations are applied. Mathematically speaking, suppose we want to modify an initial qubit state $|\psi\rangle$ using the unitary matrices A, B, and C producing a new state $|\psi'\rangle$. This is accomplished through a series of multiplications: $ABC|\psi\rangle = |\psi'\rangle$. $|\psi\rangle$ is recovered by multiplying in reverse order the inverse of each of the matrices by the new state: $C^{-1}B^{-1}A^{-1}|\psi'\rangle = |\psi\rangle$. This property of reversibility comes from the well-known result in linear algebra that the inverse of a product of invertible matrices is simply the product of the inverses of each matrix in reverse order [70]. This can be easily demonstrated using the fact that any matrix multiplied by its inverse is the identity matrix: $ABCC^{-1}B^{-1}A^{-1} = ABIB^{-1}C^{-1} = AIA^{-1} = I$.

Since all quantum operators must be unitary, there exists an inverse for every operator and that inverse is the adjoint of the operator. Thus, by keeping track of the operations performed on a set of qubits, any quantum computation can be reversed by applying the adjoint of each operation in reverse order.

An example of a commonly used operator in quantum computing is the Hadamard operator which has the form

$$H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}. \tag{1.2}$$

This operator is frequently used to put a qubit into an equal superposition (as described in Postulate 1). To illustrate, we can transform a qubit in state $|0\rangle$ into an equal superposition via matrix-vector multiplication with the Hadamard operator as follows,

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}. \tag{1.3}$$

Postulate 3. Measurement of a quantum state $|\psi\rangle$ involves a special set of operators. When such an operator Ω is applied to $|\psi\rangle$, the result will be one of the eigenvalues ω of the operator Ω with a certain probability. Measurement is destructive and will change the measured state $|\psi\rangle$ to $|\omega\rangle$.

In the context of quantum computing, this postulate has two major consequences. The first is that measuring the value of a qubit destroys its quantum state, forcing it to a discrete 0 or 1 value corresponding to classical bit states which are not superpositions of values. After measurement is performed, a quantum computation is no longer reversible in the strict Postulate 2 sense. The second consequence is that measurement is probabilistic. There are several different types of measurement, but the one that is most pertinent to this

discussion is *measurement in the computational basis*. In quantum computing, measurement in the computational basis involves measuring with respect to the $|0\rangle$ or $|1\rangle$ basis states of a qubit, forcing the qubit to a classical 0 or 1. The actual outcome (i.e. a 0 or 1 result) depends on the probability amplitude associated with each outcome in the superposition of the qubit (defined as α and β in Postulate 1). In this type of measurement, the probability of obtaining a 0 or 1 is: $p(x) = \langle\psi| M_x |\psi\rangle$ where x is either 0 or 1.[2] As an example, suppose we want to measure a quantum state $|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ in the $|1\rangle$ basis. The operator for this projective measurement is $|1\rangle\langle 1|$ which is the multiplicative product of a column vector and a row vector,

$$|1\rangle\langle 1| = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}. \tag{1.4}$$

Thus,

$$\begin{aligned} p(1) &= \langle\psi|1\rangle\langle 1|\psi\rangle \\ &= \begin{bmatrix} \alpha^* & \beta^* \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \\ &= \begin{bmatrix} 0 & \beta^* \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \\ &= |\beta|^2. \end{aligned} \tag{1.5}$$

Notice that in general when measuring in the computational basis, the probability of getting a 0 or 1 is the magnitude squared of the probability amplitude associated with that value.

[2] $\langle x|$ denotes a *bra* in the standard Dirac notation and is short-hand for the complex conjugate transpose of a complex-valued column vector (Dirac's ket) representing a quantum state.

Although ideally measurement would be performed in the computational basis to read the output at the end of a quantum computation, another form of measurement can and often does occur prematurely. This measurement comes in the form of interference from the environment surrounding the qubits and is known as *decoherence* or *quantum noise* [43, 51]. In practice it is difficult to isolate stable quantum states from the environment, and since measurement of any kind is destructive, a computation can easily be ruined before it completes. This problem alone has been one of the greatest technological hurdles facing the physical realization of quantum computers [41, 50, 51].

Postulate 4. Composite quantum states are represented by the tensor product of the component quantum states, and operators that act on composite states are represented by the tensor product of their component matrices.

Simply put, this postulate enables the description of multiple qubits and multi-qubit operators via a single state vector and matrix, respectively. The tensor product[3] is a standard linear algebraic operation. Given two matrices (vectors) A and B of dimensions $M_A \times N_A$ and $M_B \times N_B$, respectively, the tensor product $A \otimes B$ multiplies each element of A by the entire matrix (vector) B to produce a new matrix (vector) of dimensions $M_A \cdot M_B \times N_A \cdot N_B$. To illustrate, suppose we want to compute the tensor product of the following complex-valued matrices,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}. \tag{1.6}$$

[3]The tensor product is also known as the Kronecker or direct product.

The tensor product operation $\otimes$ gives

$$A \otimes B = \begin{bmatrix} ae & af & be & bf \\ ag & ah & bg & bf \\ ce & cf & de & df \\ cg & ch & dg & dh \end{bmatrix}. \tag{1.7}$$

Similarly, consider two complex-valued vectors V and W,

$$V = \begin{bmatrix} a \\ b \end{bmatrix}, \quad W = \begin{bmatrix} c \\ d \end{bmatrix}. \tag{1.8}$$

The tensor product $V \otimes W$ is

$$V \otimes W = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}. \tag{1.9}$$

In general, there is no restriction on the dimensions of tensor product operands. Matrices of different dimensions can be tensored together, as can vectors and matrices. However, in the quantum domain, we typically perform the tensor product on square, power-of-two-sized matrices to create larger operators (Postulate 2), and also on power-of-two-sized vectors to create larger composite quantum states (Postulate 1). To illustrate, suppose we want the state vector that describes the composite state of the following set of qubits: $|1\rangle, |0\rangle, |1\rangle$. The composite state vector is computed as

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}. \tag{1.10}$$

Dirac notation offers a simple short-hand description of composite quantum states in which the state symbols are simply placed side-by-side within a single ket. For the preceding example (Equation 1.10), the Dirac form is $|101\rangle$.

Extending the concept of measurement (Postulate 3) to composite quantum states is fairly straightforward. In the case of vectors, by multiplying each element of a vector V by an entire vector W, the tensor product produces a vector whose elements are indexed in binary counting order. To demonstrate, we revisit $V \otimes W$ annotated with binary indices,

$$V \otimes W = \begin{bmatrix} a \\ b \end{bmatrix}_0 \otimes \begin{bmatrix} c \\ d \end{bmatrix}_0 = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}_{\begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix}}. \tag{1.11}$$

Whereas the indices 0 and 1 for the single quantum state vectors represent the amount of “zeroness” and “oneness” in the quantum state, the indices in the above composite vec-

tor represent the amount of "00-ness, 01-ness, 10-ness, and 11-ness" respectively. Thus, when measuring with respect to the computational basis, the binary indices of a state vector denote the classical bit values that will be measured with a probability given by the magnitude squared of the value at that location in the vector.

To illustrate the construction of composite quantum operators, we turn to an example involving the Hadamard operator. A Hadamard operator that can be applied to two qubits is constructed via the tensor product of two Hadamard matrices,

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix}. \tag{1.12}$$

The above examples show that n qubits can be represented by $n-1$ tensor products of single qubit vectors, and operators that act on n qubits can be represented by $n-1$ tensor products of single qubit operators. A key point to note is that the size of a state vector resulting from a series of tensor products on n single qubit vectors is 2^n. Similarly, a composite operator which can be applied to n qubits is a matrix of size 2^{2n}. It is indeed Postulate 4 which gives rise to the exponential complexity of simulation of quantum behavior on classical computers. A straightforward linear algebraic approach to such simulation would have time and memory complexity $O(2^{2n})$ for an n-qubit system.

The No-Cloning Theorem

Another interesting property of quantum states is that they cannot be arbitrarily copied [51]. This leads to yet another fundamental difference between quantum and classical computing. In classical logic circuits, a wire can fan out from the output of a gate and feed

into many other gates. This is not possible in the quantum domain for an arbitrary qubit. However, this is not a limitation because quantum states that are known to be orthogonal to each other (including the computational basis states) can be copied. A proof adapted from [51] is offered below:

Given two unknown quantum states $|\psi\rangle$ and $|\varphi\rangle$, we try to apply some unitary operator (in accordance with Postulate 2) such that both $|\psi\rangle$ and $|\varphi\rangle$ are copied to other quantum states $|s\rangle$ and $|t\rangle$. This is represented mathematically as

$$U(|\psi\rangle \otimes |s\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{1.13}$$

$$U(|\varphi\rangle \otimes |t\rangle) = |\varphi\rangle \otimes |\varphi\rangle. \tag{1.14}$$

However, since quantum computing is modeled by a finite-dimensional Hilbert space, the inner product of both equations must be defined if they are in fact valid evolutions of quantum states. The inner product of the above two equations reduces to

$$\langle\psi|\varphi\rangle = (\langle\psi|\varphi\rangle)^2. \tag{1.15}$$

Any expression of the form $x = x^2$ (as is the case above) only has two solutions, $x = 0$ and $x = 1$. If the inner product of two state vectors is 0, the vectors are orthogonal. Also, the only way for the inner product to be 1 is if both state vectors are equal. Thus, it is either the case that $|\psi\rangle$ and $|\varphi\rangle$ are orthogonal or that $|\psi\rangle = |\varphi\rangle$. This proof demonstrates that arbitrary quantum states cannot be copied. However, if it is known that the quantum states are orthogonal, they can be copied.

The implication for quantum computing is that the computational basis states $|0\rangle$ and $|1\rangle$, which are orthogonal, *can be copied*. Since the computational basis states are anal-

ogous to the classical bit values 0 and 1, the no-cloning theorem suggests that quantum computers are at least as powerful as classical computers.

A standard quantum operator used to copy computational basis states (among other functions) is called the *CNOT* operator [51]. As the name implies, CNOT is a controlled-NOT operation. It is a unitary matrix (in accordance with Postulate 2) that acts on two qubits. One qubit is the *control qubit* while the other qubit is the *target qubit*. When the control qubit is in the $|1\rangle$ state, the CNOT is "activated", and the state of the target qubit is flipped from $|0\rangle$ to $|1\rangle$ or vice-versa. If the control qubit is in the $|0\rangle$ state, however, the state of the target qubit is unchanged. When both the control and target qubits are in the computational basis states, the CNOT operation performs the same function as the classical XOR gate where the target qubit receives the value of the XOR of the control qubit and the old target qubit value. To demonstrate, a CNOT operator is shown below changing the state vector $|10\rangle$ to $|11\rangle$,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \tag{1.16}$$

An extension of CNOT is the *Toffoli* operator, which is basically a CNOT with two control qubits and one target qubit. In this case, the value of the target qubit is flipped if *both* of the control qubits are in state $|1\rangle$. So, given two control qubits *a* and *b* and a target qubit *c*, the Toffoli gate causes *c* to become $c \oplus ab$. The Toffoli gate alone is a universal gate set that can effect any form of classical computation [51]. This is easily demonstrated by showing that the Toffoli gate can perform the same function as the classical NAND gate

which is known to be a universal gate set. To compute *a* NAND *b* where *a* and *b* are input qubits, we simply make *a* and *b* the control qubits of the Toffoli gate and initialize the target qubit to $|1\rangle$. Such an instance of the Toffoli gate computes the function $1 \oplus ab = \neg(ab)$, which is equivalent to *a* NAND *b*.

Entanglement

Yet another interesting property of quantum states is *entanglement*. Two quantum systems are entangled if the measurement outcome of one system affects the measurement statistics of another system, without any physical interaction at the time of measurement. A simple example of entangled states is the Bell state or EPR pair [51]. Suppose two parties, Alice and Bob, each have their own qubit, and the state of both qubits together is given as, $\psi_{AB} = |0_A 0_B\rangle$, where the subscript A denotes the portion of the state due to Alice's qubit, and the subscript B denotes the portion due to Bob's qubit. An EPR pair can be generated from this state by applying a Hadamard gate and a CNOT gate as follows,

$$\psi_{EPR} = (CNOT)(H \otimes I)\,|0_A 0_B\rangle = \frac{1}{\sqrt{2}}(|0_A 0_B\rangle + |1_A 1_B\rangle). \tag{1.17}$$

The utility of this state lies in the fact that if Alice measures her particle and obtains a 0, then Bob will subsequently also obtain a 0 upon measurement of his particle (the same holds true for a measurement of 1). Once the EPR pair is created, the measurement outcomes of each qubit are correlated even if Alice and Bob physically separate their qubits by any amount of distance. As a result, entanglement has applications in quantum teleportation [8] and secure public key exchange [6, 7, 27], which will be discussed later in Chapters IV and VI.

The Density Matrix Representation

An important extension of the state vector is the *density matrix*. For the purposes of quantum circuit simulation, it is sufficient to define an *n*-qubit density matrix as $\rho = |\psi\rangle\langle\psi|$, where $|\psi\rangle$ is a single state vector for a sequence of *n* initialized qubits, and $\langle\psi|$ is its complex-conjugate transpose. In other words, ρ is a $2^n \times 2^n$ matrix constructed by multiplying a 2^n element column vector with a 2^n element row vector. This operation is also known as the *outer product*. To illustrate, when $|\psi\rangle$ is a single qubit,

$$\rho = |\psi\rangle\langle\psi| = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} [\alpha^* \beta^*] = \begin{bmatrix} \alpha\alpha^* & \alpha\beta^* \\ \beta\alpha^* & \beta\beta^* \end{bmatrix} \tag{1.18}$$

Like the state vector model, a gate operation U can be applied to a density matrix, but it takes the form $U\rho U^\dagger$, where $U^\dagger$ is the complex-conjugate transpose of the matrix for U. For example, if $U = H$, the Hadamard operator,

$$H\rho H^\dagger = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} \alpha\alpha^* & \alpha\beta^* \\ \beta\alpha^* & \beta\beta^* \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \tag{1.19}$$

Perhaps the most useful property of the density matrix is that it can accurately represent a subset of the qubits in a circuit. One can extract this subset of information with the *partial trace* operation, which produces a smaller matrix, called the *reduced* density matrix [51]. To understand how this extraction can be done, consider the following example in which a 1-qubit operator U is applied to two qubits $|\psi\rangle$ and $|\phi\rangle$. The density matrix version of this circuit is $(U \otimes U)|\psi\phi\rangle\langle\psi\phi|(U \otimes U)^\dagger = |\psi'\phi'\rangle\langle\psi'\phi'|$. The state of $|\phi\rangle$ alone after U is applied, for instance, can be extracted with the partial trace, $tr(U|\psi\rangle\langle\psi|U^\dagger)U|\phi\rangle\langle\phi|U^\dagger$. tr is the standard trace operation, which produces a single complex number that is the sum of the diagonal elements of a matrix. A more concrete example is the partial trace over

the first qubit in a density matrix representing two qubits with the state $\rho_0 \otimes \rho_1$, such that $\rho_0 = |+\rangle\langle+|$ and $\rho_1 = |0\rangle\langle 0|$, where $|+\rangle$ denotes an equal superposition.

$$\rho_0 \otimes \rho_1 = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{1.20}$$

$$tr_{\rho_0}(\rho_0 \otimes \rho_1) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{1.21}$$

Although in this example the partial trace reproduces the state of the second qubit, it does not always extract the original state of the remaining qubits. In particular, when entanglement exists among two or more qubits, the partial trace will not undo the tensor product. This issue is central to some of the simulation methods discussed in Chapter II.

Also notice that the partial trace "traces over" the qubit that is *not wanted*, leaving behind the desired qubit states. Using the partial trace to extract information about subsets of qubits in a circuit is invaluable in simulation. As will be shown in Chapter IV, many practical quantum circuits contain ancillary qubits which help to perform an intermediate function in the circuit but contain no useful information at the output of the circuit. The partial trace therefore allows a simulation to report the density matrix information only for the qubits that contain useful data. Another application of the partial trace in quantum circuits is the modeling of noise from the environment. Coupling between the environment and data qubits can be modeled as the tensor product of data qubits with quantum states controlled by the environment [51]. In such a situation, the partial trace can be used to extract the state of data qubits after being affected by noise. For these reasons and others,

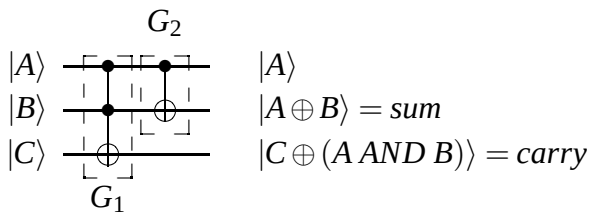


Figure 1.1: Reversible quantum half-adder circuit.

it is crucial that a quantum simulator support the density matrix representation.

1.2.2 Quantum Circuits

With the quantum mechanical background in place, we proceed to the topic of quantum circuits. Quantum circuits are analogous to the logic design level of classical computation, and therefore in this work we model all quantum computation at the quantum circuit level. The two major components in a quantum circuit are the qubits (Postulate 1) and the operators or gates (Postulate 2). The values of the qubits are observed through measurement (Postulate 3), and multiple qubits and gates can be expressed via the tensor product (Postulate 4). Clearly, the postulates of quantum mechanics provide a complete set of properties with which to perform logic design subject to the fanout constraint of the no-cloning theorem. In the remainder of this subsection, we cover two small quantum circuit examples to familiarize the reader with the standard quantum circuit notation.

The first example, shown in Figure 1.1, is a quantum half-adder. It performs the same function as the standard half-adder in classical logic circuits when the inputs are all in the computational basis. Notice that the qubits are depicted graphically as parallel, horizontal lines. These lines can be thought of as wires, but more abstractly they represent the evolution of the qubits over time. Gates are depicted as objects placed on top of the

horizontal qubit lines, affecting only those qubits lines that they are in contact with graphically, similar to a classical logic gate. The spacing between gates on the qubit lines has no significance. The only important aspect of gate placement is whether one gate appears before another, implying an order of operations to be performed on the affected qubits. The quantum half-adder simply consists of a Toffoli gate (G_1) affecting all three qubits followed by a CNOT gate (G_2) affecting the first two qubits only. The solid circles represent inputs for the control qubits, while the unfilled circles represent inputs/outputs for the target qubits. In general, the input qubits are placed at the left end of the qubit lines, with the final output state of the qubits appearing at the right end. The matrix representation of the half-adder gates is written as,

(1.22)

$$H\_Adder = G_2G_1 = (C \otimes I)T = \left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

An interesting thing to note is that although the circuit diagram flows in a left to right fashion (e.g. G_1 is applied before G_2), the matrices representing the unitary operators are applied in a seemingly reverse order. As shown above, the matrix for G_2 (i.e. $CNOT \otimes I$; the identity matrix I is applied to qubit lines with no gates acting on them) appears to the left of the matrix for G_1. The reason that the matrices appear in a reverse order when

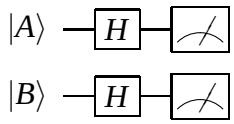


Figure 1.2: Quantum circuit which places two qubits into an equal superposition when $|A\rangle$ and $|B\rangle$ are initialized to $|0\rangle$.

read from left to right is due to the mechanics of the matrix-vector multiplication [70]. In order to perform the multiplication correctly, the state vector is multiplied on the right-hand side of an operator. Thus, in linear algebraic terms, the right to left order of the matrix representations is equivalent to the left to right order in the circuit diagram. This is analogous to a formula of the form $f(g(x))$, where the function $f(\cdot)$ is applied after $g(\cdot)$ even though $f(\cdot)$ appears before $g(\cdot)$ in the formula.

Another pertinent notational nuance is that the graphical symbols for Toffoli and CNOT gates are a bit different from those of other quantum operators. The convention for quantum gates without controls is to represent their target qubit portions as boxes containing a label describing the function of the gate. As illustrated in Figure 1.2, the Hadamard gates have no control qubits and their graphical depiction is simply a box containing an "H". It is also a convention to place meter symbols on qubit lines where measurement occurs. In Figure 1.2, the meters at the end of the qubit lines denote that measurement takes place at the end of the quantum computation.

The Toffoli and CNOT gates also play an important role in universal quantum gate sets. Analogous to their classical digital counterparts, universal quantum gate sets can be used to implement any quantum computation. However, *discrete* universal quantum gate sets (i.e. the set contains a finite number of gates) can only approximate arbitrary quantum com-

putations, though the approximation can achieve any desired level of accuracy [51]. One example of a discrete universal quantum gate set consists of the Hadamard, phase, CNOT, and $\pi/8$ gates (see Section 2.5 for a description of the phase and $\pi/8$ gates). Another discrete universal quantum gate set consists of the Hadamard, phase, CNOT, and Toffoli gates [51]. By contrast, universal quantum gate sets containing an infinite number of gates enable an exact decomposition of any quantum computation. One such gate set consists of the CNOT gate and the infinite set of all 1-qubit unitary operators [51]. Interestingly, given a circuit consisting of gates from this infinite set, the Solovay-Kitaev theorem proves that an approximation with accuracy ε can be achieved using the aforementioned discrete gate sets with only polylogarithmically more gates in terms of the number of CNOTs in the original circuit and ε [42, 51]. Thus, discrete universal gate sets are likely to be of practical value.

1.2.3 Binary Decision Diagrams

The binary decision diagram (BDD) was introduced by Lee in 1959 [45] in the context of classical logic circuit design. This data structure represents a Boolean function $f(x_1, x_2, ..., x_n)$ by a directed acyclic graph (DAG); see Figure 1.3. By convention, the top node of a BDD is labeled with the name of the function f represented by the BDD. Each variable x_i of f is associated with one or more nodes with two outgoing edges labeled *then* (solid line) and *else* (dashed line). The *then* edge of node x_i denotes an assignment of logic 1 to the x_i, while the *else* edge represents an assignment of logic 0. These nodes are called *internal* nodes and are labeled by the corresponding variable x_i. The edges of the BDD point downward, implying a top-down assignment of values to the Boolean variables

depicted by the internal nodes.

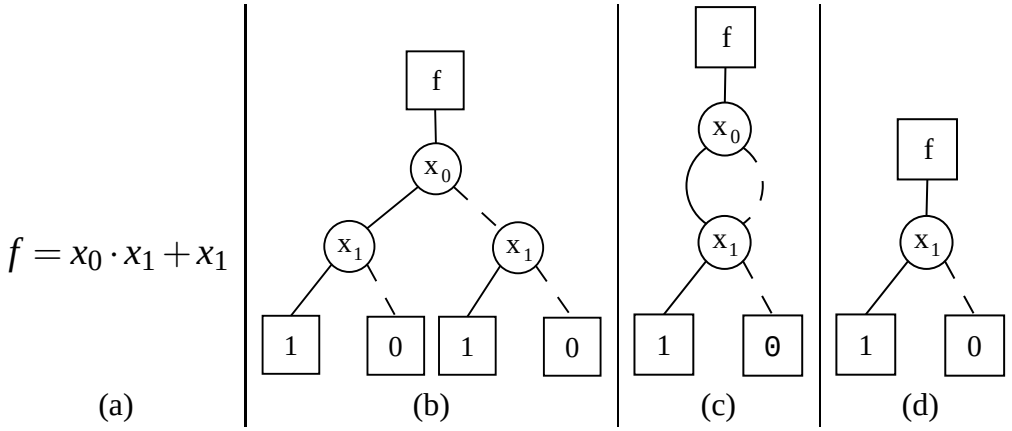


Figure 1.3: (a) A logic function, (b) its BDD representation, (c) its BDD representation after applying the first reduction rule, and (d) its ROBDD representation.

At the bottom of the BDD are *terminal* nodes containing the logic values 1 or 0. They denote the output value of the function f for a given assignment of its variables. Each path through the BDD from top to bottom represents a specific assignment of 0-1 values to the variables $x_1, x_2, \ldots, x_n$ of f, and ends with the corresponding output value $f(x_1, x_2, \ldots, x_n)$.

The original BDD data structure conceived by Lee has exponential memory complexity $\Theta(2^n)$, where n is the number of Boolean variables in a given logic function. The reason for this complexity bound is that in Lee's initial design, the paths representing all 2^n combinations of variable assignments are explicitly represented. Moreover, exponential memory and runtime are required in many practical cases, making this data structure impractical for simulation of large logic circuits. To address this limitation, Bryant developed the reduced ordered BDD (ROBDD) [17], where all variables are ordered, and decisions are made in that order. A key advantage of the ROBDD is that variable-ordering facilitates an efficient implementation of reduction rules that automatically eliminate redundancy from the basic BDD representation and may be summarized as follows:

Rule 1. There are no nodes v and v' such that the subgraphs rooted at v and v' are isomorphic

Rule 2. There are no internal nodes with *then* and *else* edges that both point to the same node

An example of how the rules transform a BDD into an ROBDD is shown in Figure 1.3. The subgraphs rooted at the x_1 nodes in Figure 1.3b are isomorphic. By applying the first reduction rule, the BDD in Figure 1.3b is converted into the BDD in Figure 1.3c. Notice that in this new BDD, the *then* and *else* edges of the x_0 node now point to the same node. Applying the second reduction rule eliminates the x_0 node, producing the ROBDD in Figure 1.3d. Intuitively it makes sense to eliminate the x_0 node since the output of the original function is determined solely by the value of x_1. An important aspect of redundancy elimination is the sensitivity of ROBDD size to the variable ordering. Finding the optimal variable ordering is an *NP*-complete problem, but efficient ordering heuristics have been developed for specific applications. Moreover, it turns out that many practical logic functions have ROBDD representations that are polynomial (or even linear) in the number of input variables [17]. In addition, the reduction rules make ROBDDs canonical, which means that no two ROBDDs represent equivalent Boolean functions. Thus, equivalence of ROBDDs can be checked in $O(1)$ time by simply comparing the root nodes [17]. Consequently, ROBDDs have become indispensable tools in the design, simulation, and synthesis of classical logic circuits.

1.2.4 BDD Operations

Even though the ROBDD is often quite compact, efficient algorithms are necessary to make it practical for circuit simulation. Thus, in addition to the foregoing reduction rules, Bryant introduced a variety of ROBDD operations whose complexities are bounded by the size of the ROBDDs being manipulated [17]. Of central importance is the **Apply** operation, which performs a binary operation with two ROBDDs, producing a third ROBDD as the result. It can be used, for example, to compute the logical *AND* of two functions. **Apply** is implemented by a recursive traversal of the two ROBDD operands. For each pair of nodes visited during the traversal, an internal node is added to the resultant ROBDD using the three rules depicted in Figure 1.4. To understand the rules, some notation must be introduced. Let v_f denote an arbitrary node in an ROBDD f. If v_f is an internal node, $Var(v_f)$ is the Boolean variable represented by v_f, $T(v_f)$ is the node reached when traversing the *then* edge of v_f, and $E(v_f)$ is the node reached when traversing the *else* edge of v_f.

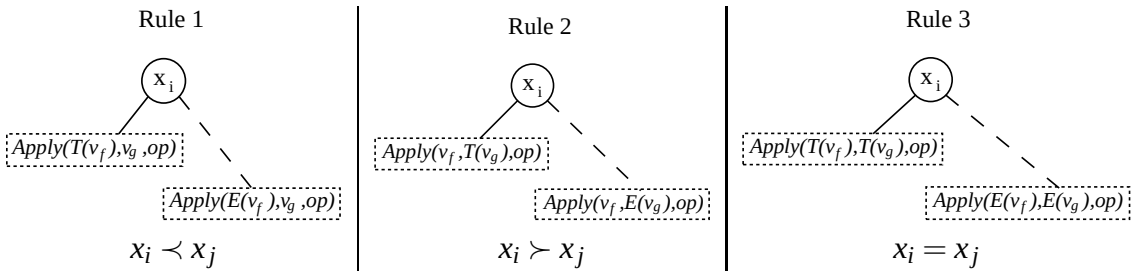


Figure 1.4: The three recursive rules used by the **Apply** operation which determine how a new node should be added to a resultant ROBDD. In the figure, $x_i = Var(v_f)$ and $x_j = Var(v_g)$. The notation $x_i \prec x_j$ is defined to mean that x_i precedes x_j in the variable ordering.

Clearly the rules depend on the variable ordering. To illustrate, consider performing

Apply using a binary operation op and two ROBDDs f and g. **Apply** takes as arguments two nodes, one from f and one from g, and the operation op. This is denoted as **Apply**(v_f, v_g, op). **Apply** compares $Var(v_f)$ and $Var(v_g)$ and adds a new internal node to the ROBDD result using the three rules. The rules also guide **Apply**'s traversal of the *then* and *else* edges (this is the recursive step). For example, suppose **Apply**(v_f, v_g, op) is called and $Var(v_f) \prec Var(v_g)$. Rule 1 is invoked, causing an internal node containing $Var(v_f)$ to be added to the resulting ROBDD. Rule 1 then directs the **Apply** operation to call itself recursively with **Apply**$(T(v_f), v_g, op)$ and **Apply**$(E(v_f), v_g, op)$. Rules 2 and 3 dictate similar actions but handle the cases when $Var(v_f) \succ Var(v_g)$ and $Var(v_f) = Var(v_g)$. To recurse over both ROBDD operands correctly, the initial call to **Apply** must be **Apply**$(Root(f), Root(g), op)$ where $Root(f)$ and $Root(g)$ are the root nodes for the ROBDDs f and g.

The recursion stops when both v_f and v_g are terminal nodes. When this occurs, op is performed with the values of the terminals as operands, and the resulting value is added to the ROBDD result as a terminal node. For example, if v_f contains the value logical 1, v_g contains the value logical 0, and op is defined to be $\oplus$ (XOR), then a new terminal with value $1 \oplus 0 = 1$ is added to the ROBDD result. Terminal nodes are considered *after* all variables are considered. Thus, when a terminal node is compared to an internal node, either Rule 1 or Rule 2 will be invoked depending on which ROBDD the internal node is from. The pseudo-code for **Apply** is provided in Figure 1.5 (the unary version is very similar).

The success of ROBDDs in making a seemingly difficult computational problem tractable

```
Apply(A, B, b_op) {
  if (Is_Constant(A) and Is_Constant(B)) {
    return New_Terminal(b_op(Value(A),
      Value(B)));
  }
  if (Table_Lookup(R, b_op, A, B)) return R;
  v = Top_Var(A, B);
  T = Apply(A_v, B_v, b_op);
  E = Apply(A_v', B_v', b_op);
  R = ITE(v, T, E);
  Table_Insert(R, b_op, A, B);
  return R;
}
```

Figure 1.5: Pseudo-code for the **Apply** algorithm. *Top_Var* returns the variable index from either A or B that appears earlier in the ordering, while ITE creates a new internal node with children T and E.

in practice led to the development of ROBDD variants outside the domain of logic design. Of particular relevance to this work are multi-terminal binary decision diagrams (MTBDDs) [22] and algebraic decision diagrams (ADDs) [4]. These data structures are compressed representations of matrices and vectors rather than logic functions, and the amount of compression achieved is proportional to the frequency of repeated values in a given matrix or vector. Additionally, some standard linear-algebraic operations, such as matrix multiplication, are defined for MTBDDs and ADDs. Since they are based on the **Apply** operation, the efficiency of these operations is proportional to the size in nodes of the MTBDDs or ADDs being manipulated. Further discussion of the MTBDD and ADD representations is deferred to Chapter III where the general structure of the QuIDD is described.

1.3 Motivation for Simulation

Building on the background information about quantum mechanics, the quantum circuit model, and decision diagrams, we now turn to the need for quantum circuit simulation techniques. Interest has recently grown in efficient, classical simulation of quantum circuits for a variety of reasons. As noted earlier, quantum circuit simulation provides an experimental testbed for the development of quantum error correction [87, 46], enables synthesis and verification of new quantum circuits [56, 61], and probes the boundaries between quantum and classical computation [84, 37, 80, 76, 1, 72, 48].

Each of these applications has different requirements. To simulate errors, it is important that the simulation technique is not limited to subsets of states and operators. As will be discussed in Chapter VI, continuous error effects originating from gate imprecision and decoherence cause each matrix element of the states and operators to be different in general. Some simulation techniques overcome this issue by keeping states separated as long as there is little or no entanglement [37, 84].

Synthesis and verification require representations of states and operators that efficiently expose certain equivalence properties. As noted in the last section, ROBDDs are canonical, which allows exact equivalence of classical bit states to be checked in $O(1)$ time by comparing head nodes only. In Chapter V, we show how this property and others are exploited by QuIDDs to improve equivalence checking in the quantum domain.

Probing the boundaries between quantum and classical computation can be approached in many ways. Although entanglement has been shown to be a necessary condition for quantum computation to achieve asymptotic improvements over classical computation

[37, 84], it is not a sufficient condition since certain quantum operators exist which generate large amounts of entanglement but exist in finite groups whose size does not increase with the number of qubits [31, 1]. Other interesting linear-algebraic properties such as the matrix Pfaffian have also been exploited to prove insufficiency for other sets of quantum operators [72]. Whether or not such techniques are practical for error simulation or synthesis, the only requirement for probing the quantum and classical computational boundaries is to prove that sub-exponential time and memory complexities exist for classical simulation of some class of quantum states and/or operators.

In the next chapter, we delve into the details of many of these quantum circuit simulation techniques as well as a few others not listed here. Each technique exploits one or more peculiar properties of classical representations of quantum states and operators. The chapters that follow this survey discuss in detail the QuIDD-based technique that we have developed and explain how QuIDDs fit in with all of the major simulation applications just discussed.

1.4 Thesis Outline

QuIDDs are discussed in detail in Chapter III, including a formal description of a practical class of quantum states and operators which can be simulated efficiently using QuIDDs. Quantum search is used as a benchmark in Chapter III to evaluate the effectiveness of QuIDDs. The results indicate that QuIDDs enable efficient simulation of two common instances of quantum search as well as a useful class of quantum states and operators. Further details on this class are provided in Appendix A.

We have also developed significant extensions to the QuIDD data structure which en-

able efficient simulation with density matrices, which is a very useful simulation model for incorporating error effects [76]. These extensions are described in Chapter IV. Also, density matrix-based simulation with QuIDDs is compared to NIST's QCSim simulator on a number of quantum circuit benchmarks including error correction, reversible logic, quantum communication, and quantum search. Our experimental data demonstrates that QuIDDs significantly outperform QCSim on all benchmarks.

Chapter V addresses the the goal of verifying synthesized quantum circuits by simulating such circuits and checking for equivalence among the resultant states and operators. Although checking exact equality for both states and operators is a very efficient operation with QuIDDs, quantum information introduces other notions of equivalence due to global and relative phases. Various linear-algebraic and QuIDD properties may be exploited to check various conditions for such equivalences. A number of QuIDD algorithms implementing these checks is described and analyzed in Chapter V. Results for a number of benchmarks show that the QuIDD algorithms enable fast equivalence checking in practical cases.

Throughout the work, various attributes of the QuIDDPro simulator are discussed. This software tool implements the QuIDD data structure and all related algorithms with a robust, expressive front-end language. Far from being a set of implementation details, we show in Chapter VI that the front-end language enables some automatic speed-up techniques for QuIDD-based simulation. In addition to making QuIDDs competitive with techniques like the stabilizer formalism (see Section 2.5), we leverage these speed-ups to accurately characterize the effects of gate, systematic, and decoherence error in a quantum

circuit that generates remotely entangled EPR pairs. “Bang-bang” error correction is also simulated in this circuit, confirming its effectiveness in combating decoherence error.

Appendix B provides a brief overview of the QuIDDPro simulator and the complete QuIDDPro language reference. Appendix C offers several QuIDDPro scripts which simulate some of the quantum circuits discussed in this dissertation as well as a few well-known quantum states. These scripts illustrate how the QuIDDPro language is both compact and expressive. Chapter VII summarizes the contributions of this dissertation and discusses a few perspectives on future applications to related problems.

CHAPTER II

Survey of Simulation Techniques

In this chapter we survey the major methods proposed for quantum circuit simulation methods. In particular, we discuss qubit-wise multiplication, *p*-blocked simulation, tensor networks, Vidal's slightly entangled technique, the stabilizer formalism, and a few other techniques. Most of these methods simulate specific classes of quantum circuits efficiently without approximation.

In addition to these simulation methods, a number of "programming environments" for quantum computing were proposed recently [53, 54, 18] that are mostly front-ends to quantum circuit simulation techniques. This distinction between front-end (language and development environment) and back-end (key algorithms and simulation engine) is similar to what is commonly found in classical circuit simulation. Many of these programming environments use naive quantum circuit simulation back-ends which explicitly multiply matrices and require super-polynomial computational resources in the number of qubits. Although choosing to interface to such a back-end may ease the job of the front-end developer, the potential benefits of efficient linear-algebraic operations on compressed arguments are immense. To illustrate the benefits a more efficient technique would offer,

consider a 20-qubit system. Such a system entails a $2^{20} \times 2^{20}$ complex-valued matrix, whose storage is well beyond the memory available in modern computers.

Traditional array-based representations are often insensitive to the actual values stored, and even sparse matrix storage offers little improvement for quantum operators with no zero matrix elements (e.g. Hadamard operators). However, the techniques described here are more sophisticated, and in this chapter we examine their advantages and disadvantages.

2.1 Qubit-wise Multiplication

One popular array-based simulation technique is to simulate *k*-input quantum gates on an *n*-qubit state-vector ($k \leq n$) without explicitly storing a $2^n \times 2^n$-matrix representation [12, 52]. The basic idea is to simulate the full-fledged matrix-vector multiplication by a series of simpler operations. To illustrate, consider simulating a quantum circuit in which a 1-qubit Hadamard operator is applied to the third qubit of the state-space $|00100\rangle$. The state-vector representing this state-space has 2^5 elements. A naive way to apply the 1-qubit Hadamard to $|00100\rangle$ is to construct a $2^5 \times 2^5$ matrix of the form $I \otimes I \otimes H \otimes I \otimes I$ and then multiply this matrix by the state vector. However, rather than compute $(I \otimes I \otimes H \otimes I \otimes I)|00100\rangle$, one can simply compute $|00\rangle \otimes H|1\rangle \otimes |00\rangle$, which produces the same result using a 2×2 matrix H. The same technique can be applied when the state-space is in a superposition, such as $\alpha|00100\rangle + \beta|00000\rangle$. In this case, to simulate the application of a 1-qubit Hadamard operator to the third qubit, one can compute $|00\rangle \otimes H(\alpha|1\rangle + \beta|0\rangle) \otimes |00\rangle$. As in the previous case, a 2×2 matrix is sufficient.

While the above method allows one to compute a state space symbolically, in a realistic simulation environment state vectors may be much more complicated. Shortcuts

that take advantage of the linearity of matrix-vector multiplication are desirable. For example, a single qubit can be manipulated in a state vector by extracting a certain set of two-dimensional vectors. Each vector in such a set is composed of two probability amplitudes. The corresponding qubit states for these amplitudes differ in value at the position of the qubit being operated on, but agree in every other qubit position. The two-dimensional vectors are then multiplied by matrices representing single qubit gates in the circuit being simulated. We refer to this technique as *qubit-wise multiplication* because the state-space is manipulated one qubit at a time. Obenland implemented a technique of this kind as part of a simulator for quantum circuits [52]. His method applies one- and two-qubit operator matrices to state vectors of size 2^n. Unfortunately, in the best case where $k = 1$, this only reduces the runtime and memory complexity from $O(2^{2n})$ to $O(2^n)$, which is still exponential in the number of qubits.

Another implicit limitation of Obenland's implementation is that it simulates with the state-vector representation only. The qubit-wise technique has been extended, however, to enable density matrix simulation by Black et al. and is implemented in NIST's QCSim simulator [12]. As in its predecessor simulators, the arrays representing density matrices in QCSim tend to grow exponentially. This asymptotic bottleneck is demonstrated experimentally in Sections 3.3 and 4.4.

2.2 P-blocked Simulation

Avoiding the exponential complexity of the state vector or density matrix can be achieved by keeping the state separated, if possible, in pieces that do not grow exponentially in size. To track separable states, it is more typical to use metrics of entanglement and develop

state representations whose size is sensitive to such metrics. To this end, Jozsa and Linden offer a p-blocked state representation, which can be used to simulate any quantum circuit with low p [37]. This algorithm decomposes the state into blocks of entangled qubits up to size p, where no $p+1$ qubits are entangled, $\rho = \rho_1 \otimes \rho_2 \otimes \cdots \otimes \rho_k$, where ρ_i is the density matrix for qubit i. Since each block requires at least 2^p coefficients, the space complexity grows with the number of entangled qubits.

Unfortunately, applying 2-qubit operators that straddle different blocks requires combining both blocks via the tensor product. Once the operator is applied, all possible partial traces which break up the combined block into two smaller blocks (combinatorially many in general) are taken. If the tensor product of any two smaller blocks equals the combined block, then the smaller blocks become the updated blocks representing that portion of the state. If all possible partial traces fail to produce such blocks, then the combined block becomes part of the updated representation, increasing p for the system.

Although it may be possible to perform fewer partial traces with proper analysis of a given circuit, a more significant drawback is that for commonly used states such as $|\psi_{\text{cat}}\rangle = (|00\ldots0\rangle + |11\ldots1\rangle)/\sqrt{2}$ (the "cat" or GHZ state), this representation requires exponential space in n when $p \gg \log(n)$. For example, consider the 2-qubit cat state, which is an EPR pair created by the circuit described in Equation 1.17. Computing the partial trace over both qubits produces two density matrices whose tensor product is not equal to the density matrix of the original state as shown below.

$$(2.1) \qquad tr_{|\psi_A\rangle\langle\psi_A|}(|\psi_{EPR}\rangle\langle\psi_{EPR}|) = tr\left(\begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \end{bmatrix}\right) = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix}$$

$$(2.2) \qquad \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \otimes \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} = \begin{bmatrix} 1/4 & 0 & 0 & 0 \\ 0 & 1/4 & 0 & 0 \\ 0 & 0 & 1/4 & 0 \\ 0 & 0 & 0 & 1/4 \end{bmatrix} \neq |\psi_{EPR}\rangle\langle\psi_{EPR}|$$

As a result, p must be increased upon creation of the EPR pair, and p increases exponentially with the number of qubits entangled in this fashion when creating an n-qubit cat state. In contrast, QuIDDs can represent $|\psi_{\text{cat}}\rangle$ using $O(n)$ space by exploiting the massive redundancy in the amplitudes of this state, as will be demonstrated in Chapters III and IV.

2.3 Tensor Networks

While qubit-wise multiplication targets operators only and p-blocked simulation targets separable states, tensor networks capture structure in quantum circuits that allow efficient simulation. Markov and Shi develop this approach by making use of graphs of tensors, which are a multi-dimensional generalization of matrices [48]. In this method, tensors represent density matrix states and operators. For example, the operator U acting on a input qubits and b output qubits is denoted as follows [48],

$$(2.3) \qquad [U_{\sigma_1,\sigma_2,\ldots,\sigma_a,\tau_1,\tau_2,\ldots,\tau_b}]_{\sigma_1,\sigma_2,\ldots,\sigma_a,\tau_1,\tau_2,\ldots,\tau_b},$$

where each $\sigma_i, \tau_j \in |b_1\rangle\langle b_2| : b_1, b_2 \in \{0,1\}$. Here each index can take on one of the four

possible index values of a 1-qubit density matrix.

A separate tensor is created for each gate in a circuit. Treated like a node in a graph, each tensor is connected to other tensors via shared qubit indices (input/output connections). These graphs or *tensor networks* make use of an operation called *tensor contraction* which merges connected nodes containing tensors into a single tensor. Tensor contraction is simply the multi-dimensional generalization of the dot product. To illustrate, consider the tensor contraction of tensor g and h over a shared output/input connection [48]:

$$f_{i_1,\ldots,i_m,j'_1,\ldots,j'_{n'}} = \sum_{j_1,\ldots,j_n} g_{i_1,\ldots,i_m,j_1,\ldots,j_n} \cdot h_{j_1,\ldots,j_n,j'_1,\ldots,j'_{n'}}. \tag{2.4}$$

The goal of this method is to contract all tensors into a single tensor describing the action of the circuit on qubits of interest. Depending on how the tensors are connected, contractions may either decrease, increase, or leave unaffected the dimensions of the resultant tensor as compared to the dimensions of the separate tensors. To illustrate, consider the tensor contraction of F and G over the shared index o as shown in Figure 2.1. F and G are tensor representations of two 2-qubit quantum gates where an output wire (output tensor index) o of F is an input wire (input tensor index) to G. Notice that in this case, tensor contraction produces a new tensor H with larger dimensions than F and G.

Simulation with this method is exponential in d, the maximum dimension of any tensor created by contractions. The tensor-network approach is also applicable to instances of one-way quantum computation [16]. However, tensor networks can be insensitive to separable states, and therefore in practice are combined with other simulation techniques [64]. Aharonov et al. extend tensor networks to demonstrate that the quantum Fourier transform (QFT), a key operation in Shor's integer factoring algorithm [65], can be simu-

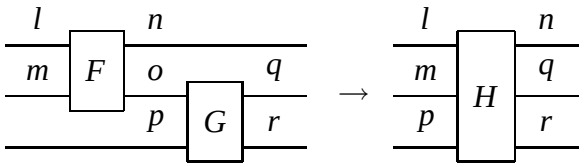


Figure 2.1: Tensor contraction of shared wire (index) o for tensors F and G, each of which represents a 2-qubit gate.

lated efficiently on a classical computer [2]. The result does not enable efficient number factoring on a classical computer, however, since a different operation called modular exponentiation remains a bottleneck.

2.4 Slightly Entangled Simulation

While p-blocked simulation separates states via tensor products only, more sophisticated techniques may be used to exploit state separability even further. Vidal offers one such technique which utilizes the Schmidt decomposition (SD) of the quantum state [84]. Cat states and separable states are represented with only quadratic overhead by Vidal's technique [84]. Consider n qubits ordered from 0 to $n-1$. A *bipartite splitting* $A:B$ of the qubits is given by any integer k from $1..n-2$ in the sense that qubits $i \leq k$ and $j > k$ form two complementary partitions A and B, respectively. Then the state $|\psi\rangle$ can be decomposed as follows [23],

$$|\psi\rangle = \sum_{\alpha=0}^{\chi_A - 1} \lambda_\alpha \left|\Phi_\alpha^{[A]}\right\rangle \otimes \left|\Phi_\alpha^{[B]}\right\rangle. \tag{2.5}$$

Here, $\left|\Phi_\alpha^{[A]}\right\rangle$ and $\left|\Phi_\alpha^{[B]}\right\rangle$ are two orthonormal bases, and $\sum_\alpha |\lambda_\alpha|^2 = 1$. It is common to take $\left|\Phi_\alpha^{[A]}\right\rangle$ and $\left|\Phi_\alpha^{[B]}\right\rangle$ as eigenvectors of the reduced density matrices $\rho^{[A]}$ and $\rho^{[B]}$, respectively, which both have the same eigenvalue $|\lambda_\alpha|^2 > 0$. The *Schmidt rank* χ_A is a

measure of entanglement between partitions A and B, and each of χ_A addends consists of two vector terms. The entanglement of state $|\psi\rangle$ can be quantified by the maximum χ_A over all possible bipartite splittings $A:B$, $\chi \equiv \max_A \chi_A$ [84, Equation 2]. Depending on the amount of entanglement, χ can range from 1 for fully separable states, to 2^n for fully entangled states.

The space complexity of Vidal's representation and the time complexities of related algorithms are functions of χ. In particular, Vidal decomposes an n-qubit state into sums of tensor products [84, Equation 15] that we refer to as a *dense tensor decomposition* (DTED),

$$|\psi\rangle = \sum_{\alpha_0=0,\cdots,\alpha_{n-2}=0}^{(\chi_0-1),\cdots,(\chi_{n-2}-1)} \left|\varphi_{\alpha_0}^{[0]}\right\rangle \lambda_{\alpha_0}^{[0]} \left|\varphi_{\alpha_0\alpha_1}^{[1]}\right\rangle \ldots \lambda_{\alpha_{n-2}}^{[n-2]} \left|\varphi_{\alpha_{n-2}}^{[n-1]}\right\rangle, \tag{2.6}$$

In this equation, vectors $\left|\varphi_{\alpha_{l-1}\alpha_l}^{[l]}\right\rangle$ are unnormalized 1-qubit states, and the Schmidt coefficients $\lambda_{\alpha_l}^{[l]}$ express the correlation information between qubits $0..l$ and qubits $(l+1)..(n-1)$ (the tensor product symbols are omitted for simplicity). Each α_l index may range from 0 to $\chi_l - 1$. The DTED of a pure n-qubit state $|\psi\rangle$ is derived by applying the SD $n-1$ times to the bipartite splittings $0:n-1, 1:n-1, \ldots, n-2:n-1$, in such a way that the maximal possible rank is χ. Each time, this process generates χ_l coefficients $\lambda_{\alpha_l}^{[l]}$ and χ_l^2 2-element vectors $\left|\varphi_{\alpha_{l-1}\alpha_l}^{[l]}\right\rangle$. Therefore, the DTED decomposes $|\psi\rangle$ into a sum of up to χ^n separable states and requires $n(2\chi^2+\chi)$ complex-valued coefficients [84].

To simulate 1-qubit gates and 2-qubit gates, one uses algorithms that update the DTED state representation. Vidal offers algorithms that take $O(\chi^2)$ time for 1-qubit gates and $O(\chi^3 + n\chi^2)$ time[1] for nearest-neighbor 2-qubit gates [84]. For a generic circuit with

[1]When a 2-qubit operator is applied to qubits l and $l+1$, partial traces over all other qubits must be

g gates, Vidal's protocol runs in $O(ng\chi^3 + n^2 g\chi^2)$ time. In particular, applying 2-qubit operators to qubits l and $l+1$ requires solving a potentially large eigenvalue problem to update $\lambda_{\alpha_l}^{[l]}$ [84]. While the precise complexity of measurement is not given in [84], we believe that it requires $O(n\chi^2)$ time in the DTED formalism.[2]

To see how these complexity results are derived, consider the algorithms for 1- and 2-qubit operator updates in Vidal's DTED representation. A 1-qubit unitary operator U is applied to qubit l as follows [84, Equation 16],

$$\left|\varphi'^{[l]}_{\alpha_{l-1}\alpha_l}\right\rangle = U\left|\varphi^{[l]}_{\alpha_{l-1}\alpha_l}\right\rangle \forall \alpha_l, \alpha_{l+1} = 0, \cdots, (\chi-1). \tag{2.7}$$

This operation takes $O(\chi^2)$ time since $\alpha_{l-1}, \alpha_l \leq \chi$. DTED updates for 2-qubit operators applied to qubits l and $l+1$ are much more involved. Vidal explicitly solves for the eigenvalues and eigenvectors of $\rho^{[(l+1)..(n-1)]}$ (see Equation 2.5), which requires several major steps. The first step is to apply the 2-qubit unitary operator V to the substates corresponding to qubits l and $l+1$ in the following way [84, Equation 22],

$$\Theta^{[l,(l+1)]}_{\alpha_{l-1}\alpha_{l+1}} = \sum_{\alpha_l} V\left|\varphi^{[l]}_{\alpha_{l-1}\alpha_l}\right\rangle \lambda^{[l]}_{\alpha_l}\left|\varphi^{[l+1]}_{\alpha_l\alpha_{l+1}}\right\rangle. \tag{2.8}$$

The resultant density matrix of the second partition becomes [84, Equation 23],

(2.9)

$$\rho'^{[(l+1)\cdot\cdot(n-1)]} = \sum_{j,j',\alpha_{l+1},\alpha'_{l+1}} \Big(\sum_{\alpha_{l-1}} \langle\alpha_{l-1}|\alpha_{l-1}\rangle \Theta^{[l,(l+1)]}_{\alpha_{l-1}\alpha_{l+1}} (\Theta^{[l,(l+1)]}_{\alpha_{l-1}\alpha'_{l+1}})^*\Big) |j\alpha_{l+1}\rangle \langle j'\alpha'_{l+1}|,$$

where $j = 0, 1$, and [84, Equations 18 and 19],

computed, requiring $O(n\chi^2)$ time for this step [84, Equations 13, 14, 18, 19, 23, and 26]. This term is not included in [84, Lemma 2] because it is dominated by χ^3 when $\chi \gg n$. However, it can be signifi cant for slightly entangled states which are the focus of [84].

[2] Vidal notes that measurement can be accomplished in time polynomial in χ but apparently assumes in the analysis that $\chi = \Omega(n)$.

$$|\alpha_{l-1}\rangle \equiv \lambda_{\alpha_{l-1}}^{[l-1]} \left|\Phi_{\alpha_{l-1}}^{[0\cdots(l-1)]}\right\rangle, \tag{2.10}$$

$$|\alpha_{l+1}\rangle \equiv \lambda_{\alpha_{l+1}}^{[l+1]} \left|\Phi_{\alpha_{l+1}}^{[(l+2)\cdots(n-1)]}\right\rangle. \tag{2.11}$$

Computing $\langle\alpha_{l-1}|\alpha_{l-1}\rangle$ using Equation 2.10 requires $O(n\chi^2)$ time [84, Equation 13]. Equation 2.8 is computed using $O(\chi^3)$ time since there are three consecutive α indices, each of which is bounded by χ. With $\langle\alpha_{l-1}|\alpha_{l-1}\rangle$ and Equation 2.8 computed, Equation 2.9 is computed using $O(\chi^3)$ time since it involves summing over all combinations of α_{l+1}, α'_{l+1}, and α_{l-1}. The new Schmidt coefficients $\lambda'^{[l]}_{\alpha_l}$ are generated by solving for the eigenvalues of $\rho'^{[(l+1)\cdots(n-1)]}$, which can be done using $O(\chi^3)$ time. The new states $\left|\varphi^{[l+1]}_{\alpha_l\alpha_{l+1}}\right\rangle$ are computed by decomposing the eigenvalues and eigenvectors in terms of $|j\alpha_{l+1}\rangle$ using $O(\chi)$ time [84, Equation 24]. Lastly, the new states $\left|\varphi^{[l]}_{\alpha_{l-1}\alpha_l}\right\rangle$ are computed by decomposing the eigenvalues, eigenvectors, and $\langle\alpha_{l+1}|\alpha_{l+1}\rangle$ terms with respect to $|\alpha_{l-1}i\rangle$, where $i = 0, 1$, requiring $O(\chi)$ time [84, Equations 26 and 27]. The overall time complexity of the 2-qubit operator update is therefore $O(\chi^3)$ while increasing α_l by up to 2χ.

To illustrate how Vidal's protocol works, consider again the creation of an EPR pair. A simplified version of the notation is used below to track Vidal's algorithms after application of the Hadamard and CNOT gates. Initially, the states are unentangled since both qubits have the value $|0\rangle$. This means that $\chi = 1$, $\left|\Phi_0^{[0]}\right\rangle = \left|\Phi_0^{[1]}\right\rangle = |0\rangle$, and $\lambda_0 = 1$.

$$H|0\rangle \otimes |0\rangle = |+\rangle \otimes |0\rangle \tag{2.12}$$

$$CNOT(|+\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle) \quad + \quad \frac{1}{\sqrt{2}}(|1\rangle + |1\rangle) \tag{2.13}$$

The Hadamard gate does not increase χ since it is applied only to the first qubit. The CNOT gate, however, increases χ by one as indicated by the presence of a second tensor product term in the summation. The CNOT gate is applied by computing the tensor product $|+\rangle \otimes |0\rangle$, multiplying the resulting 4-element vector by the matrix representing CNOT, computing the density matrix of the resulting vector via the outer product, and solving for both the eigenvalues and eigenvectors of this density matrix. λ contains the square roots of the two eigenvalues shared by the reduced density matrices of each qubit (reduced via the partial trace), and the new state vectors are the eigenvectors of the reduced density matrices.

An important drawback of DTED and Vidal's simulation protocol is the redundancy in state encoding. For a generic state with maximum entanglement ($\chi = 2^n$), DTED requires $\Omega(n2^{2n})$ coefficients, whereas 2^n amplitudes suffice to characterize the state. Interestingly, *p*-blocked simulation and QuIDDs represent generic, maximally entangled states using only $O(2^n)$ space [37, 80]. A key open question is whether the extra coefficients are necessary to ensure that space and time complexity of quantum simulation remain polynomial in χ.

2.5 Stabilizer Circuit Formalism

The techniques described so far offer general-purpose quantum circuit simulation. However, sacrificing generality by focusing on particular subsets of quantum operators can lead to further improvements in simulation. Gottesman describes a simulation method involving the *Heisenberg representation* of quantum computation which tracks the commutators of a particular group of operators applied in a quantum circuit [31]. With this

model, the state need not be represented explicitly by a state-vector or a density matrix because the operators describe how an arbitrary state-vector would be altered by the circuit. Gottesman shows that simulation based on this model requires only polynomial memory and runtime on a classical computer in certain cases. However, efficient simulation with this method is limited to quantum circuits containing operators in the Clifford group. These operators do not form a universal gate set. A recent extension to this technique enables simulation with any quantum operators, but the complexity grows exponentially with every operator introduced that is not a generator of the Clifford group [1].

To illustrate how this technique works, we revisit the quantum circuit described in Section 1.2.1, which generates an EPR pair (Equation 1.17). Initially, the two qubits are in the ground state $|00\rangle$. In general, *n stabilizers* are needed to represent an *n*-qubit state [1]. A stabilizer[3] is an *n*-qubit operator composed of the tensor product of Pauli matrices, which are members of the Clifford group. The Pauli matrices perform rotations of qubit state vectors on the X, Y, and Z axes and can be described as

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{2.14}$$

Representing the initial state $|00\rangle$ requires two stabilizers, namely $Z \otimes I$ and $I \otimes Z$. The choice of these particular stabilizers is not arbitrary and is derived from the following equation which converts stabilizers to the density matrix of an arbitrary *n*-qubit quantum state $|\Psi\rangle$ [1],

[3]The term stabilizer is generally given to any operator with respect to any state vector that is not altered by application of the operator to all qubits in the state.

$$|\Psi\rangle\langle\Psi| = \frac{1}{2^n}\Pi_{i=1}^{n}(I+M_i), \tag{2.15}$$

where the M_i are the stabilizers. Setting $M_1 = Z \otimes I$ and $M_2 = I \otimes Z$ verifies that this choice of stabilizers generates the desired initial state. Applying a gate U to the state is defined by applying U to each stabilizer M_i via the matrix multiplication $UM_iU^\dagger$. For the stabilizer formalism to be efficient, the stabilizers and operations on them cannot be represented with explicit matrices and matrix operations. Such efficiency can be accomplished by representing the stabilizers with Pauli symbols (e.g. "ZI" and "IZ") and by applying transformations through a set of rules which map Pauli symbols to other Pauli symbols. This is possible due to a basic result in group theory. Specifically, when a member of a group is applied to another member of a group, the result is just another member of the group, possibly multiplied by factors of -1 and i. By restricting the gates of a quantum circuit to members of the Clifford group, it is guaranteed that the stabilizers representing the quantum state will always be members of the Clifford group. In fact, it is equivalent to restrict the allowable gates to the generators of the Clifford group since any member of a group can be reconstructed using the group's generators. The generators of the Clifford group are the CNOT gate, the Hadamard gate, and the phase gate [31]. The phase gate S is represented by the following matrix,

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}. \tag{2.16}$$

Multiplication of explicit matrices is not required to update the state after applying a gate that is a Clifford generator. Instead, a simple look-up table can be employed which con-

tains the transformation rules for applying Clifford group generators to Pauli matrices [31, 1, 51]. These rules are given in Table 2.1.

Gate	Input	Output
H	X	Z
	Z	X
S	X	Y
	Z	Z
$CNOT$	$X \otimes I$	$X \otimes X$
	$I \otimes$	$I \otimes X$
	$Z \otimes I$	$Z \otimes I$
	$I \otimes Z$	$Z \otimes Z$

Table 2.1: Transformation rules for applying Clifford group generators to Pauli operators [31, 51]. Each transformation rule is equivalent to the expression $Output = Gate * Input * Gate^{\dagger}$. Some transformations are not shown explicitly since they can be generated by combinations of the transformations listed. For instance, Y is equivalent to $SXS^{\dagger}$.

Returning to the EPR pair example, the initial state is represented by the symbols "ZI" and "IZ." Using the transformation rules shown in Table 2.1 and the rule that applying any gate to the identity operator returns the identity operator, the Hadamard and CNOT transformations are given as follows.

$$ZI, IZ \rightarrow_H XI, IZ \tag{2.17}$$

$$XI, IZ \rightarrow_{CNOT} XX, ZZ \tag{2.18}$$

Plugging the final stabilizers "XX" and "ZZ" into Equation 2.15 confirms that they do indeed represent the correct final state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle$.

Interestingly, the states produced by the stabilizer formalism represent a limited set of probabilistic outcomes for the qubits. In particular, the probability of obtaining a $|0\rangle$ or $|1\rangle$ upon measurement of any qubit in such a state is always either 0, 1 or $1/2$ [1]. Determining

the measurement probability for any qubit along with the transformation on the stabilizers induced by the measurement outcome can be easily accomplished using rules similar to those shown in Table 2.1 [1].

As noted earlier, the stabilizer formalism can be extended to incorporate gates outside of the Clifford group. It is easily shown, for example, that any 1-qubit operator can be decomposed into a sum of Pauli operators as follows,

$$U = \Sigma_j c_j P_j, \tag{2.19}$$

where c_j is a complex-valued coefficient and P_j is a Pauli matrix. As a result, the simple transformation rules can be applied to the stabilizer symbols as before, with the only difference being that each stabilizer may potentially become a sum of different Pauli operators. For example, consider the "pi/8" gate whose matrix T is

$$T = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix} = \alpha Z + \beta I, \tag{2.20}$$

where $\alpha = 0.1464 - 0.3568i$ and $\beta = 1 - \alpha$. Suppose T is applied to a single qubit whose stabilizer is "X." By applying the stabilizer transformation rules and rearranging terms, $TXT^\dagger$ produces $(X+Y)/\sqrt{2}$. The new state can be represented with two stabilizer symbols "X" and "Y", and one or two coefficients. In the worst-case, each application of a non-Clifford group operator doubles the number of stabilizer symbols and coefficients that must be maintained per stabilizer, leading to asymptotically exponential runtime and memory usage.

Aside from this potential blow-up, the formalism still provides a convenient representation for many quantum circuits since each stabilizer symbol in a sum can be updated with

the same rules. The only change is that when performing measurement, each set of stabilizer symbols in a sum contributes a probability of measurement of 0, 1 or $1/2$ multiplied by the appropriate coefficient $|c_j|^2$.

A simple and efficient implementation of the stabilizer formalism involves updating a table of bits [51, 1]. Since there are four Pauli matrices, two bits are required per Pauli operator symbol in a stabilizer. Formally, two binary variables, x_{ij} and z_{ij} are assigned to each Pauli operator in a stabilizer. By convention, $x_{ij} = 1$ indicates the presence of an X, whereas $z_{ij} = 1$ indicates the presence of a Z. A value of 1 for both bits indicates the presence of a Y, whereas a value of 0 for both bits indicates the presence of an I [51]. For an n-qubit circuit, the table of bits is represented as,

$$\left[\begin{array}{ccc|ccc} x_{11} & \dots & x_{1n} & z_{11} & \dots & z_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nn} & z_{n1} & \dots & z_{nn} \end{array}\right]. \tag{2.21}$$

Each row represents one stabilizer, so the dimensions of the table are $n \times 2n$. In the EPR example considered above, the initial ground state is represented by the stabilizers "ZI" and "IZ" which in tabular form is

$$\left[\begin{array}{cc|cc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]. \tag{2.22}$$

According to Table 2.1, applying a Hadamard gate to the first qubit transforms a "Z" to an "X." In the table, this is easily accomplished by simply swapping the bits x_{11} and z_{11}. In general, applying a Hadamard gate to qubit j is accomplished by swapping the bits x_{ij} and z_{ij} for all i. All other transformations are accomplished by similar bit manipulations. As a result, any gate that is a Clifford group generator can be applied with runtime complexity

$O(n)$. Using a modest number of extra bits, determining the measurement outcome and modifying the stabilizers based on that outcome can be accomplished with runtime complexity $O(n^3)$ [1]. Thus, simulation of quantum circuits containing gates that are Clifford group generators requires $O(n^3)$ runtime and memory resources. For each gate applied that is outside the Clifford group, the bit table is copied and assigned a coefficient as described earlier. Each table is then modified separately using the same bit manipulation rules. In the worst case, an exponential number of tables will need to be created.

Anders and Briegel recently proposed a modified stabilizer simulation technique which uses graphs instead of bit tables [3]. While the size complexity of the bit table is $O(n^2)$, the size complexity of the graph representation is $O(nlog(n))$. This more compact representation also reduces the time complexity of applying gates and performing measurements.

2.6 Other Simulation Techniques

This section briefly mentions a few other techniques which so far have found less practical applicability in the field, but nevertheless involve other interesting properties. Valiant provides a technique which can efficiently simulate yet another class of quantum circuits whose properties are based on efficiently computing the Pfaffian of relevant matrices [72], where the Pfaffian is a mathematical construct resembling the determinant of a matrix. However, computing the Pfaffian will not be efficient in general for arbitrary gates.

Other advanced simulation techniques including MATLAB's "packed" representation, apply data compression to matrices and vectors, but cannot perform matrix-vector multiplication without first decompressing the matrices and vectors. A notable exception is Greve's graph-based simulation of Shor's algorithm which uses BDDs [32]. Probability

amplitudes of individual qubits are modeled by single decision nodes. Unfortunately, this only captures superpositions where every participating qubit is rotated by ± 45 degrees from $|0\rangle$ toward $|1\rangle$.

2.7 Summary

This chapter described a number of sophisticated techniques for quantum circuit simulation. Each technique exploits a particular property of quantum states and/or operators. Qubit-wise multiplication, for example, exponentially reduces the complexity of storing operators but maintains the state vector or density matrix explicitly.

Both p-blocked simulation and Vidal's technique exploit separable states, which are states with small amounts of entanglement. In the p-blocked method, states are separated into tensor products of density matrices with size $O(2^{2p})$. Gates which affect qubits in separate partitions require combining the affected partitions via the tensor product before the gate operation is applied. Since no heuristic is offered to compute the partial traces of the resultant block, the worst-case is assumed which is combinatorial in the number of qubits. More importantly, tensor products alone do not compress many forms of entanglement, making p an overestimate of the level of entanglement in the system. Vidal improves this measure by using the maximal Schmidt rank χ of the state. His method additionally provides a systematic way to break up blocks of multiple qubits by solving eigenvalue problems and expressing the state with sums of tensor products. An interesting open question is whether or not time and memory complexities based on χ are less efficient in the worst-case situation of maximal entanglement. Vidal's method also requires $O(n)$ swaps in arbitrary quantum circuits since only nearest-neighbor qubits may be manipulated with

gates.

Tensor networks offer an alternative way of compressing quantum circuits by systematically contracting tensors representing neighboring gates. Quantum circuits whose tensor network representation has low treewidth (a graph-based measure of the maximally-sized tensor created by contractions) are simulated efficiently. In general, treewidth is not sensitive to entanglement, but tensor networks are readily combined with techniques that are sensitive. The stabilizer formalism exploits a finite group of operators to efficiently simulate a class of quantum circuits that contain only those operators. The stabilizer formalism is extremely fast in practice for such circuits since simulation reduces to the manipulation of bit tables via fixed transformation rules.

In the remaining chapters we discuss our QuIDD simulation technique. Theoretical as well as practical properties of the QuIDD technique are analyzed for a number of quantum circuit CAD applications.

CHAPTER III

State Vector Simulation with QuIDDs

This chapter is based on material appearing in [79, 80, 82, 83]. We have developed practical algorithms for simulating quantum circuits on conventional computers using the state vector representation. It is implemented using a data structure we have developed called the QuIDD which uses decision diagram concepts that are well-known in the context of simulating classical computer hardware [22, 4, 17]. This chapter demonstrates that QuIDDs allow simulations of n-qubit systems to achieve run-time and memory complexities that range from $O(1)$ to $O(2^n)$, and the worst case is not typical. In the important case of Grover's quantum search algorithm [33], we show that a QuIDD-based simulator outperforms other known simulation techniques in terms of asymptotic runtime and memory usage.

3.1 QuIDD Theory

The QuIDD was born out of the observation that vectors and matrices which arise in quantum computing contain entries and sub-matrices which occur repeatedly. Complex operators obtained from the tensor product of simpler matrices continue to exhibit this type of repeated sub-structure which certain BDD variants can capture. MTBDDs and ADDs,

introduced in Section 1.2.4, are particularly relevant to the task of simulating quantum systems. The QuIDD can be viewed as an ADD or MTBDD with the following properties:

1. The values associated with terminal nodes are complex numbers.

2. Rather than contain the values explicitly, QuIDD terminal nodes contain integer indices which map into a separate array of complex numbers. This allows the use of a simpler integer function for **Apply**-based operations, along with existing ADD and MTBDD libraries [66], greatly reducing implementation overhead.

3. The variable ordering of QuIDDs interleaves row and column variables, which favors compression of repeated sub-structure

4. Bahar et al. [4] note that ADDs can be padded with 0's to represent arbitrarily sized matrices. No such padding is necessary in the quantum domain where all vectors and matrices have sizes that are a power of 2

We demonstrate using our QuIDD-based simulator QuIDDPro that these properties greatly enhance the performance of quantum computational simulation.

3.1.1 Vectors and Matrices

Figure 3.1 shows the QuIDD structure for three 2-qubit states. We consider the indices of the four vector elements to be binary numbers, and define their bits as decision variables of QuIDDs. A similar definition is used for ADDs [4]. For example, traversing the *then* edge (solid line) of node I_0 in Figure 3.1c is equivalent to assigning the value 1 to the first bit of the 2-bit vector index. Traversing the *else* edge (dotted line) of node I_1 in the same figure is equivalent to assigning the value 0 to the second bit of the index. These traversals

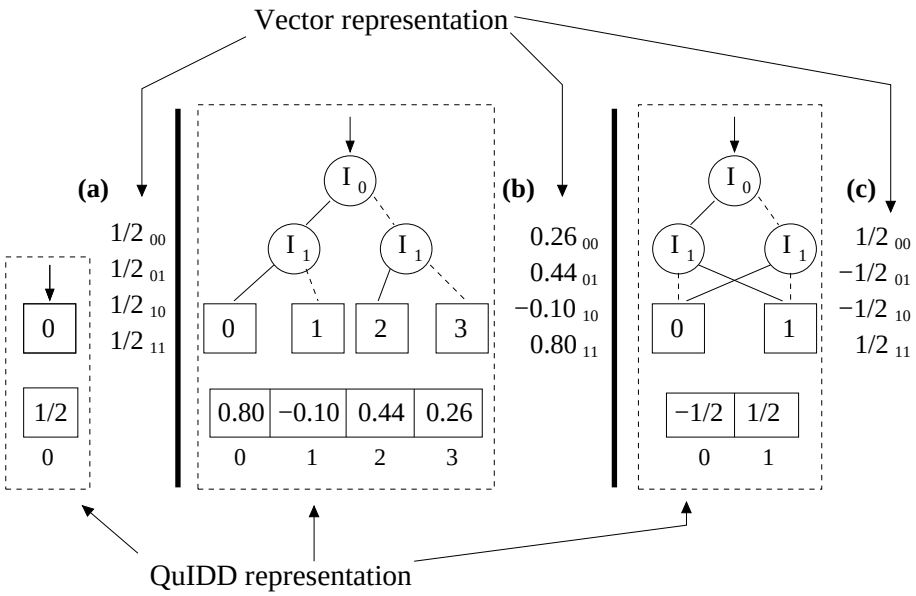


Figure 3.1: Sample QuIDDs for state vectors of (a) best, (b) worst and (c) mid-range size.

bring us to the terminal value $-\frac{1}{2}$, which is precisely the value at index 10 in the vector representation.

QuIDD representations of matrices extend those of vectors by adding a second type of variable node and enjoy the same reduction rules and compression benefits. Consider the 2-qubit Hadamard matrix annotated with binary row and column indices shown in Figure 3.2a. In this case there are two sets of indices: the first (vertical) set corresponds to the rows, while the second (horizontal) set corresponds to the columns. We assign the variable name R_i and C_i to the row and column index variables, respectively. This distinction between the two sets of variables was originally noted in several works including that of Bahar et al. [4]. Figure 3.2b shows the QuIDD form of this sample matrix where it is used to modify the state vector $|00\rangle = (1,0,0,0)$ via matrix-vector multiplication, an operation discussed in more detail in Subsection 3.1.4.

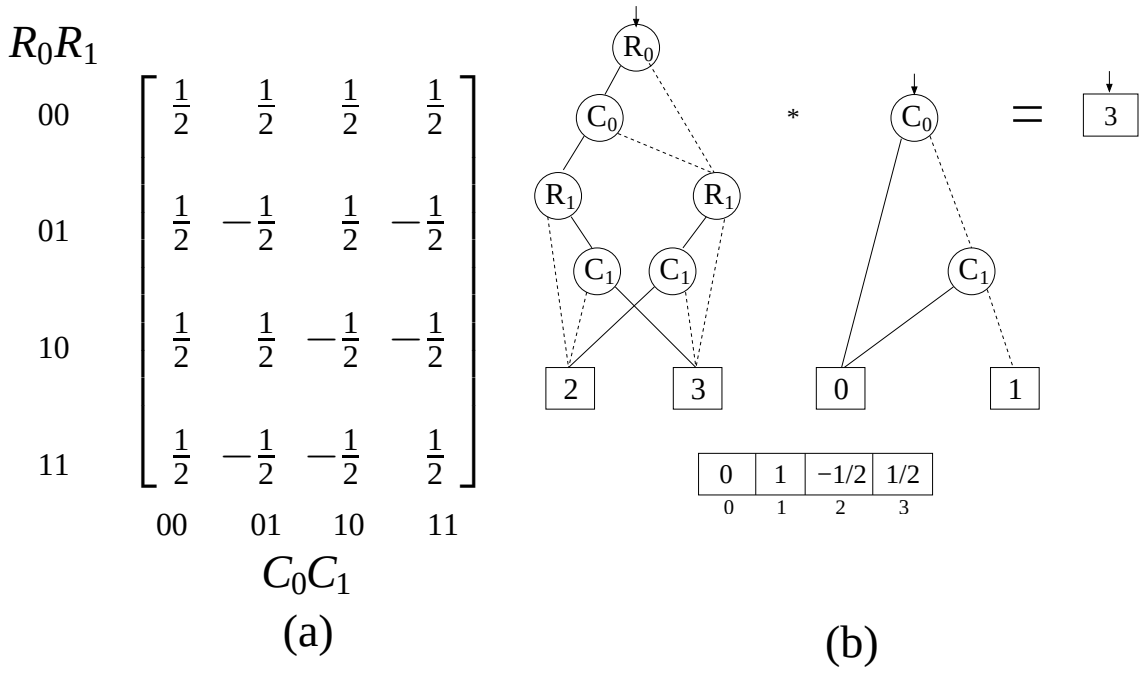


Figure 3.2: (a) 2-qubit Hadamard matrix, and (b) its QuIDD representation multiplied by $|00\rangle = (1,0,0,0)$. Note that the vector and matrix QuIDDs share the entries in a terminal array that is global to the computation.

3.1.2 Variable Ordering

As explained in Subsection 1.2.3, variable ordering can drastically affect the level of compression achieved in BDD-based structures such as QuIDDs. The CUDD programming library [66], which is incorporated into QuIDDPro, offers sophisticated dynamic variable-reordering techniques that achieve performance improvements in various BDD applications. However, dynamic variable reordering has significant time overhead, whereas finding a good static ordering in advance may be preferable in some cases. Good variable orderings are highly dependent upon the problem domain. In the case of quantum computing, we notice that all matrices and vectors contain 2^n elements where n is the number of qubits represented. Additionally, the matrices are square and non-singular [51].

McGeer et al. demonstrated that ADDs representing certain rectangular matrices can be operated on more efficiently if row and column variables are interleaved [24]. This interleaving employs the following variable ordering: $R_0 \prec C_0 \prec R_1 \prec C_1 \prec \ldots \prec R_n \prec C_n$. Intuitively, the interleaved ordering causes compression to favor regularity in particular sub-structures of the matrices that are partitions broken up into equally sized quadrants or blocks. We observe that such regularity is created by tensor products that allow multiple quantum gates to operate in parallel and also to extend smaller quantum gates to operate on larger numbers of qubits. The tensor product $A \otimes B$ multiplies each element of A by the whole matrix B to create a larger matrix which has dimensions $M_A \cdot M_B \times N_A \cdot N_B$. By definition, the tensor product propagates block patterns in its operands. To illustrate the notion of sub-structure and how QuIDDs take advantage of it, consider the tensor product of two one-qubit Hadamard operators,

(3.1)

$$\left[\begin{array}{c|c} (1/\sqrt{2}) & (1/\sqrt{2}) \\ \hline (1/\sqrt{2}) & -1/\sqrt{2} \end{array}\right] \otimes \left[\begin{array}{c|c} (1/\sqrt{2}) & (1/\sqrt{2}) \\ \hline (1/\sqrt{2}) & -1/\sqrt{2} \end{array}\right] = \left[\begin{array}{c|c} \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & -1/2 \end{pmatrix} & \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & -1/2 \end{pmatrix} \\ \hline \begin{pmatrix} 1/2 & 1/2 \\ 1/2 & -1/2 \end{pmatrix} & \begin{matrix} -1/2 & -1/2 \\ -1/2 & 1/2 \end{matrix} \end{array}\right].$$

The above matrices have been separated into quadrants, each of which represents a block. For the Hadamard matrices depicted, three of the four blocks are equal in both of the one-qubit matrices and also in the larger two-qubit matrix (the equivalent blocks are surrounded by parentheses). This repetition of equivalent blocks demonstrates that the tensor product of two equal matrices propagates block patterns. In the above example, all blocks but the lower-right block of an n-qubit Hadamard operator are equal. Furthermore, the structure of the two-qubit matrix implies a recursive sub-structure, which can be seen by recursively partitioning each of the quadrants in the two-qubit matrix,

(3.2)

$$\left[\begin{array}{c|c} (1/\sqrt{2}) & (1/\sqrt{2}) \\ \hline (1/\sqrt{2}) & -1/\sqrt{2} \end{array}\right] \otimes \left[\begin{array}{c|c} (1/\sqrt{2}) & (1/\sqrt{2}) \\ \hline (1/\sqrt{2}) & -1/\sqrt{2} \end{array}\right] = \left[\begin{array}{c|c} \left(\begin{array}{c|c} (1/2) & (1/2) \\ \hline (1/2) & -1/2 \end{array}\right) & \left(\begin{array}{c|c} (1/2) & (1/2) \\ \hline (1/2) & -1/2 \end{array}\right) \\ \hline \left(\begin{array}{c|c} (1/2) & (1/2) \\ \hline (1/2) & -1/2 \end{array}\right) & \begin{array}{c|c} (-1/2) & (-1/2) \\ \hline (-1/2) & 1/2 \end{array} \end{array}\right].$$

The only difference between the values in the two-qubit matrix and the values in the one-qubit matrices is a factor of $1/\sqrt{2}$. Thus, we can recursively define the Hadamard operator as follows,

(3.3)
$$H^{n-1} \otimes H^{n-1} = \left[\begin{array}{cc} C_1 H^{n-1} & C_1 H^{n-1} \\ C_1 H^{n-1} & C_2 H^{n-1} \end{array}\right].$$

where $C_1 = 1/\sqrt{2}$ and $C_2 = -1/\sqrt{2}$. Other operators constructed via the tensor product can also be defined recursively in a similar fashion.

Since three of the four blocks in an n-qubit Hadamard operator are equal, significant redundancy is exhibited. The interleaved variable ordering property allows a QuIDD to explicitly represent only two distinct blocks rather than four as shown in Figure 3.3. Sections 3.2 and 3.3 demonstrate that compression of equivalent blocks using QuIDDs offers major performance improvements for many of the operators that are frequently used in quantum computation. In the next subsection, we describe an algorithm which implements the tensor product for QuIDDs and leads to the compression just described.

3.1.3 Tensor Product

With the concepts of structure and variable ordering in place, operations involving QuIDDs can now be defined. Most operations defined for ADDs also work on QuIDDs

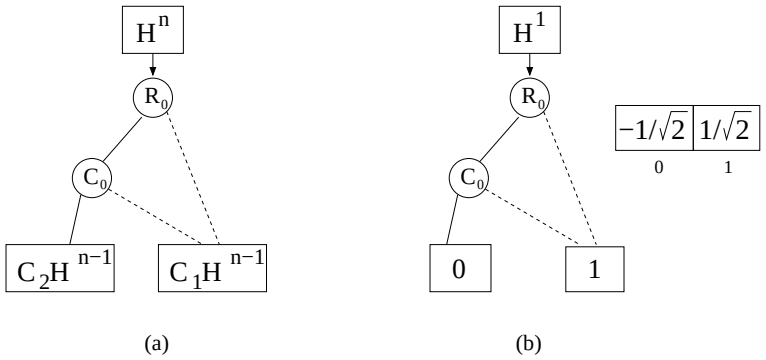


Figure 3.3: (a) *n*-qubit Hadamard QuIDD depicted next to (b) 1-qubit Hadamard QuIDD. Notice that they are isomorphic except at the terminals.

with some modification to accommodate the QuIDD properties. The tensor (Kronecker) product has been described by Clarke et al. for MTBDDs representing various arithmetic transform matrices [22]. Here we reproduce an algorithm for the tensor product of QuIDDs based on the **Apply** operation that bears similarity to Clarke's description. Recall that the tensor product $A \otimes B$ produces a new matrix which multiplies each element of A by the entire matrix B. Rows (columns) of the tensor product matrix are component-wise products of rows (columns) of the argument matrices. Therefore it is straightforward to implement the tensor product operation on QuIDDs using the **Apply** function with an argument that directs **Apply** to multiply when it reaches the terminals of both operands. The main difficulty here lies in ensuring that each terminal of A is multiplied by *all* the terminals of B. From the definition of the standard recursive **Apply** routine, we know that variables which precede other variables in the ordering are expanded first [17, 22]. Therefore, we must first shift all variables in B in the current order *after* all of the variables in A prior to the call to **Apply**. After this shift is performed, the **Apply** routine will then produce the desired behavior. **Apply** starts out with $A * B$ and expands A alone until $A_{terminal} * B$ is reached for

each terminal in A. Once a terminal of A is reached, B is fully expanded, implying that each terminal of A is multiplied by all of B.

The size of the resulting QuIDD $A \otimes B$ and the runtime for generating it given two operands A and B of sizes $|A|$ and $|B|$ (in number of nodes) is $O(|A||B|)$ because the tensor product simply involves a variable shift of complexity $O(|B|)$, followed by a call to **Apply**, which Bryant showed to have time and memory complexity $O(|A||B|)$ [17].

3.1.4 Matrix Multiplication

Matrix multiplication can be implemented very efficiently by using **Apply** to implement the dot-product operation. This follows from the observation that multiplication is a series of dot-products between the rows of one operand and the columns of the other operand. In particular, given matrices A and B with elements a_{ij} and b_{ij}, their product $C = AB$ can be computed element-wise by $c_{ij} = \Sigma_{j=1}^{n} a_{ij} b_{ji}$.

Matrix multiplication for QuIDDs is an extension of the **Apply** function that implements the dot-product. One call to **Apply** will not suffice because the dot-product requires *two* binary operations to be performed, namely addition and multiplication. To implement this, we simply use the matrix multiplication algorithm defined by Bahar et al. for ADDs [4] but modified to support the QuIDD properties. The algorithm essentially makes two calls to **Apply**, one for multiplication and the other for addition.

Another important issue in efficient matrix multiplication is compression. To avoid the same problem that MATLAB encounters with its "packed" representation, ADDs do not require decompression during matrix multiplication. Bahar et al. [4] addressed this by tracking the number i of "skipped" variables between the parent node and its child node in

each recursive call. To illustrate, suppose that $Var(v_f) = x_2$ and $Var(T(v_f)) = x_5$. In this situation, $i = 5 - 2 = 3$. A factor of 2^i is multiplied by the terminal-terminal product that is reached at the end of a recursive traversal [4].

The pseudo-code presented for this algorithm suggests time-complexity $O((|A||B|)^2)$, where A and B are two ADDs [4]. As with all algorithms based on **Apply**, the size of the resulting ADD is on the order of the time complexity, that is $O((|A||B|)^2)$. For QuIDDs, we use a modified form of this algorithm to multiply operators by the state vector, meaning that $|A|$ and $|B|$ will be the sizes in nodes of a QuIDD matrix and QuIDD state vector, respectively. If either a or b or both are exponential in the number of qubits in the circuit, the QuIDD approach will have exponential time and memory complexity. However, in Section 3.2 we prove that many of the operators which arise in quantum computing have QuIDD representations that are polynomial in the number of qubits.

Two important modifications must be made to the ADD matrix multiply algorithm in order to adapt it for QuIDDs. To satisfy QuIDD properties 1 and 2, the algorithm must treat the terminals as indices into an array rather than the actual values to be multiplied and added. Also, variable ordering must be accounted for when multiplying a matrix by a vector. A QuIDD matrix is composed of interleaved row and column variables, whereas a QuIDD vector only depends on column variables. If the ADD algorithm is run as described above without modification, the resulting QuIDD vector will be composed of row instead of column variables. The structure will be correct, but the dependence on row variables prevents the QuIDD vector from being used in future multiplications. Thus, we introduce a simple extension which transposes the row variables in the new QuIDD vector to the

corresponding column variables. In other words, for each R_i variable that exists in the QuIDD vector's support, we map that variable to C_i.

3.1.5 Other Linear-Algebraic Operations

Matrix addition is easily implemented by calling **Apply** with op defined to be addition. Unlike the tensor product, no special variable order shifting is required for matrix addition. Another interesting operation which is nearly identical to matrix addition is element-wise multiplication $c_{ij} = a_{ij}b_{ij}$. Unlike the dot-product, this operation involves only products and no summation. This algorithm is implemented just like matrix addition except that op is defined to be multiplication rather than addition. In quantum simulation, this operation is useful for matrix-vector multiplications with a diagonal matrix like the conditional phase shift in Grover's algorithm [33]. Such a shortcut considerably improves upon full-fledged matrix multiplication. Interestingly enough, element-wise multiplication and matrix addition operations for QuIDDs also implement scalar multiplication and addition without loss of efficiency. That is because a QuIDD with a single terminal node can be viewed either as a scalar value or as a matrix or vector with repeated values.

Since matrix addition, element-wise multiplication, and their scalar counterparts are nothing more than calls to **Apply**, the runtime complexity of each operation is $O(|A||B|)$. Likewise, the resulting QuIDD has memory complexity $O(|A||B|)$ [17].

Another relevant operation which can be performed on QuIDDs is the transpose. It is perhaps the simplest QuIDD operation because it is accomplished merely by swapping the row and column variables. The transpose is easily extended to the complex conjugate transpose by first performing the transpose of a QuIDD and then conjugating its terminal

values. The runtime and memory complexity of these operations is $O(a)$ where a is the size in nodes of the QuIDD undergoing a transpose.

To perform quantum measurement (see Subsection 3.1.6) one can use the inner product, which can be faster than multiplying by projection matrices and computing norms. Using the transpose, the inner product can be defined for QuIDDs. The inner product of two QuIDD vectors, e.g., $\langle A|B\rangle$, is computed by matrix multiplying the transpose of A with B. Since matrix multiplication is involved, the runtime and memory complexity of the inner product is $O((|A||B|)^2)$. Our QuIDD-based simulator QuIDDPro supports matrix multiplication, the tensor product, measurement, matrix addition, element-wise multiplication, scalar operations, the transpose, the complex conjugate transpose, and the inner product.

3.1.6 Measurement

Measurement can be defined for QuIDDs using a combination of operations. After measurement, the state vector is described by,

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}. \tag{3.4}$$

M_m is the measurement operator and can be represented by a QuIDD matrix, and the state vector $|\psi\rangle$ can be represented by a QuIDD vector. The numerator involves a QuIDD matrix multiplication. In the denominator, $M_m^\dagger$ is the complex conjugate transpose of M_m, which is also defined for QuIDDs. $M_m^\dagger M_m$ and $M_m^\dagger M_m|\psi\rangle$ are matrix multiplications. $\langle\psi|M_m^\dagger M_m|\psi\rangle$ is an inner product which produces a QuIDD with a single terminal node. Taking the square root of the value in this terminal node is straightforward. To complete the measurement, scalar division is performed with the QuIDD in the numerator and the single

terminal QuIDD in the denominator as operands.

There are two ways two compute the measurement result. The first way is inefficient and involves computing the above formula explicitly. Performing the matrix multiplication in the numerator has runtime and memory complexity $O((|A||B|)^2)$. The scalar division of the numerator by the denominator also has the same runtime and memory complexity since the denominator is a QuIDD with a single terminal node. However, computing the denominator will have runtime and memory complexity $O(|A|^{16}|B|^6)$ due to the matrix-vector multiplications and inner product. A more efficient method is to multiply the measurement operator as before, but instead of computing the denominator, two calls to **Apply** are made. The first call uses **Apply** to determine the norm of the state vector. The second call divides each terminal value by the norm. The dominating complexity of all these operation is due to matrix multiplication, resulting in a runtime and memory complexity of $O((|A||B|)^2)$ for measurement.

3.2 Complexity Analysis

In this section we prove that the QuIDD data structure can represent a large class of state vectors and operators using an amount of memory that is *linear* in the number of qubits rather than exponential. Further, we prove that the QuIDD operations required in quantum circuit simulation, i.e., matrix multiplication, the tensor product, and measurement, have both runtime and memory that is linear in the number of qubits for the same class of state vectors and operators. In addition to these complexity issues, we also analyze the runtime and memory complexity of simulating Grover's algorithm using QuIDDs.

3.2.1 Complexity of QuIDDs and QuIDD Operations

The key to analyzing the runtime and memory complexity of the QuIDD-based simulations lies in the mechanics of the tensor product. Indeed, the tensor product is the means by which quantum circuits can be represented with matrices. In the following analysis, the size of a QuIDD is represented by the number of nodes rather than actual memory consumption. Since the amount of memory used by a single QuIDD node is a constant, size in nodes is relevant for asymptotic complexity arguments. Actual memory usage in megabytes of QuIDD simulations is reported in Section 3.3.

Figure 3.4 illustrates the general form of a tensor product between two QuIDDs A and B. $In(A)$ represents the internal nodes of A, while $Term(A)$ denotes the terminal nodes. The notation for B is similar.

$In(A)$ is the root subgraph of the tensor product result because of the interleaved variable ordering defined for QuIDDs and the variable shifting operation of the tensor product (see Subsection 3.1.3). Suppose that A depends on the variables $R_0 \prec C_0 \prec \ldots \prec R_i \prec C_i$, and B depends on the variables $R_0 \prec C_0 \prec \ldots \prec R_j \prec C_j$. In performing $A \otimes B$, the variables on which B depends will be shifted to $R_{i+1} \prec C_{i+1} \prec \ldots \prec R_{k+i+1} \prec C_{k+i+1}$. The tensor product is then completed by calling $Apply(A, B, *)$. Due to the variable shift on B, Rule 1 of the **Apply** function (Subsection 1.2.4) will be used recursively after each comparison of a node from A with a node from B until the terminals of A are reached. Using Rule 1 for each of these comparisons implies that only nodes from A will be added to the result, explaining the presence of $In(A)$. Once the terminals of A are reached, Rule 2 of **Apply** will then be invoked since terminals are defined to appear last in the variable ordering.

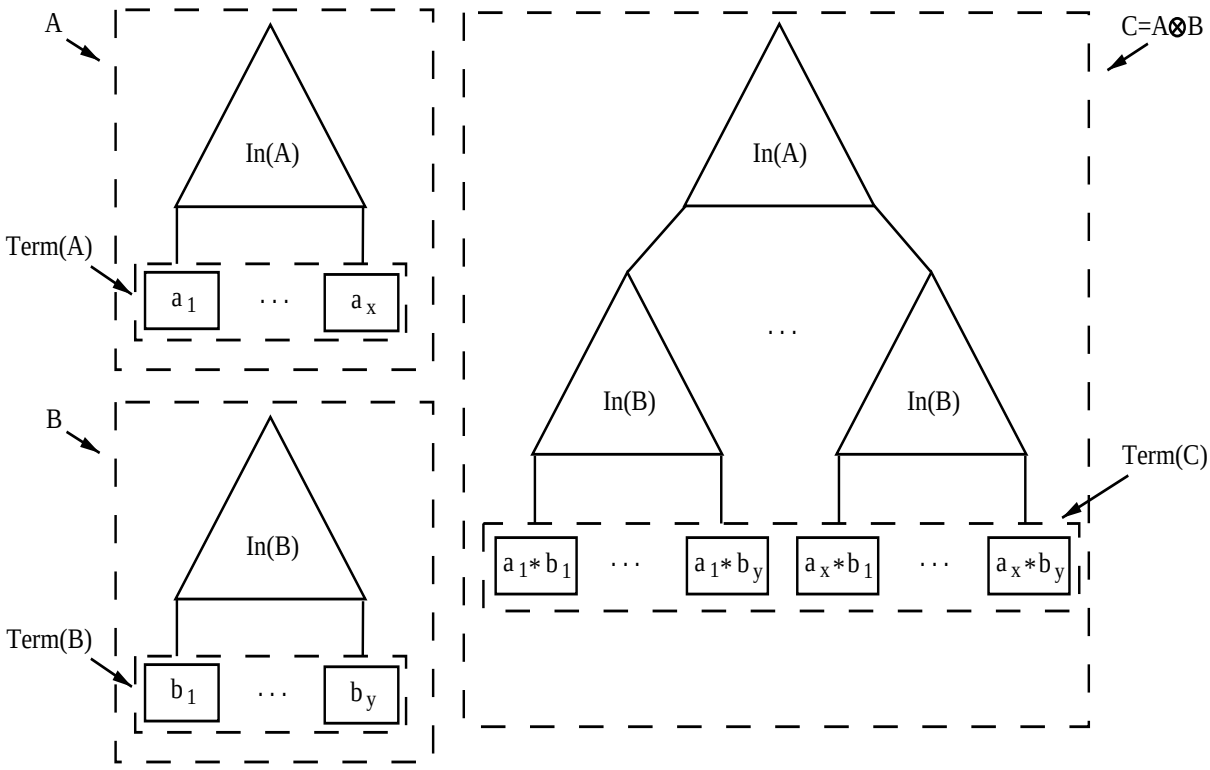


Figure 3.4: General form of a tensor product between two QuIDDs *A* and *B*.

Using Rule 2 when the terminals of *A* are reached implies that all the internal nodes from *B* will be added in place of each terminal of *A*, causing *x* copies of $In(B)$ to appear in the result (recall that there are *x* terminals in *A*). When the terminals of *B* are reached, they are multiplied by the appropriate terminals of *A*. Specifically, the terminals of a copy of *B* will each be multiplied by the terminal of *A* that its $In(B)$ replaced. The same reasoning holds for QuIDD vectors which differ in that they depend only on R_i variables.

Figure 3.4 suggests that the size of a QuIDD constructed via the tensor product depends on the number of terminals in the operands. The more terminals a left-hand tensor operand contains, the more copies of the right-hand tensor operand's internal nodes will be added to the result. More formally, consider the tensor product of a series of QuIDDs $\otimes_{i=1}^{n} Q_i =$

$(\ldots((Q_1 \otimes Q_2) \otimes Q_3) \otimes \ldots \otimes Q_n)$. Note that the $\otimes$ operation is associative (thus parenthesis do not affect the result), but it is not commutative. The number of nodes in this tensor product is described by the following lemma.

Lemma 3.5 *Given QuIDDs* $\{Q_i\}_{i=1}^n$*, the tensor-product QuIDD* $\otimes_{i=1}^n Q_i$ *contains* $|In(Q_1)| + \Sigma_{i=2}^n |In(Q_i)||Term(\otimes_{j=1}^{i-1} Q_j)| + |Term(\otimes_{i=1}^n Q_i)|$ *nodes.*[1]

Proof. This formula can be verified by induction. For the base case, $n = 1$, there is a single QuIDD Q_1. Putting this information into the formula eliminates the summation term, leaving $|In(Q_1)| + |Term(Q_1)|$ as the total number of nodes in Q_1. This is clearly correct since, by definition, a QuIDD is composed of its internal and terminal nodes. To complete the proof, we now show that if the formula is true for Q_n then it's true for Q_{n+1}. The inductive hypothesis for Q_n is $|\otimes_{i=1}^n Q_i| = |In(Q_1)| + \Sigma_{i=2}^n |In(Q_i)||Term(\otimes_{j=1}^{i-1} Q_j)| + |Term(\otimes_{i=1}^n Q_i)|$. For Q_{n+1} the number of nodes is

$$|(\otimes_{i=1}^n Q_i) \otimes Q_{n+1}| =$$

$$|\otimes_{i=1}^n Q_i| - |Term(\otimes_{i=1}^n Q_i)| + |In(Q_{n+1})||Term(\otimes_{i=1}^n Q_i)| + |Term(\otimes_{i=1}^{n+1} Q_i)|. \quad (3.6)$$

Notice that the number of terminals in $\otimes_{i=1}^n Q_i$ is subtracted from the total number of nodes in $\otimes_{i=1}^n Q_i$ and multiplied by the number of internal nodes in Q_{n+1}. The presence of these terms is due to Rule 2 of **Apply** which dictates that in the tensor-product $(\otimes_{i=1}^n Q_i) \otimes Q_{n+1}$, the terminals of $\otimes_{i=1}^n Q_i$ are replaced by copies of Q_{n+1} where each copy's terminals are multiplied by a terminal from $\otimes_{i=1}^n Q_i$. The last term simply accounts for the total number

[1] $|In(A)|$ denotes the number of internal nodes in A, while $|Term(A)|$ denotes the number of terminal nodes in A.

of terminals in the tensor-product. Substituting the inductive hypothesis made earlier for the term $|\otimes_{i=1}^{n} Q_i|$ produces

$$\begin{aligned} & |In(Q_1)| + \Sigma_{i=2}^{n} |In(Q_i)||Term(\otimes_{j=1}^{i-1} Q_j)| + |Term(\otimes_{i=1}^{n} Q_i)| - |Term(\otimes_{i=1}^{n} Q_i)| \\ & \quad + |In(Q_{n+1})||Term(\otimes_{i=1}^{n} Q_i)| + |Term(\otimes_{i=1}^{n+1} Q_i)| \\ & = |In(Q_1)| + \Sigma_{i=2}^{n+1} |In(Q_i)||Term(\otimes_{j=1}^{i-1} Q_j)| + |Term(\otimes_{i=1}^{n+1})|. \end{aligned} \tag{3.7}$$

Thus the number of nodes in Q_{n+1} is equal to the original formula we set out to prove for $n+1$ and the induction is complete. □

Lemma 3.5 suggests that if the number of terminals in $\otimes_{i=1} Q_i$ increases by a certain factor with each Q_i, then $\otimes_{i=1}^{n} Q_i$ must grow exponentially in n. If, however, the number of terminals stops changing, then $\otimes_{i=1}^{n} Q_i$ must grow linearly in n. Thus, the growth depends on matrix entries because terminals of $A \otimes B$ are products of terminal values of A by terminal values of B and repeated products are merged. If all QuIDDs Q_i have terminal values from the same set Γ, the product's terminal values are products of elements from Γ.

Definition 3.8 Consider finite non-empty sets of complex numbers Γ_1 and Γ_2, and define their *all-pairs product* as $\{xy \mid x \in \Gamma_1,\ y \in \Gamma_2\}$. One can verify that this operation is associative, and therefore the set Γ^n of *all n-element products* is well defined for $n > 0$. We then call a finite non-empty set $\Gamma \subset \mathbb{C}$ *persistent* iff the size of Γ^n is constant for all $n > 0$.

For example, the set $\Gamma = \{c, -c\}$ is persistent for any c because $\Gamma^n = \{c^n, -c^n\}$. In general any set closed under multiplication is persistent, but that is not a necessary condition. In particular, for $c \neq 0$, the persistence of Γ is equivalent to the persistence of $c\Gamma$. Another observation is that Γ is persistent if and only if $\Gamma \cup \{0\}$ is persistent. An

important example of a persistent set is the set consisting of 0 and all n-th degree roots of unity $\mathbb{U}_n = \{e^{2\pi ik/n} | k = 0..n-1\}$, for some n. Since roots of unity form a group, they are closed under multiplication and form a persistent set. In Appendix A, we show that every persistent set is either $c\mathbb{U}_n$ for some n and $c \neq 0$, or $\{0\} \cup c\mathbb{U}_n$.

The importance of persistent sets is underlined by the following theorem.

Theorem 3.9 *Given a persistent set* Γ *and a constant* C*, consider* n *QuIDDs with at most* C *nodes each and terminal values from* Γ*. The tensor product of those QuIDDs has* $O(n)$ *nodes and can be computed in* $O(n)$ *time.*

Proof. The first and the last terms of the formula in Lemma 3.5 are bounded by C and $|\Gamma|$ respectively. As the sizes of terminal sets in the middle term are bounded by $|\Gamma|$, the middle term is bounded by $|\Gamma| \sum_{i=2}^{n} |In(Q_i)| < |\Gamma|c$ since each $|In(Q_i)|$ is a constant. The tensor product operation $A \otimes B$ for QuIDDs involves a shift of variables on B followed by $Apply(A, B, *)$. If B is a QuIDD representing n qubits, then B depends on $O(n)$ variables.[2] This implies that the runtime of the variable shift is $O(n)$. Bryant proved that the asymptotic runtime and memory complexity of $Apply(A, B, binary\_op)$ is $O(|A||B|)$ [17]. Lemma 3.5 and the fact that we are considering QuIDDs with at most C nodes and terminals from a persistent set Γ imply that $|A| = O(n)$ and $|B| = O(1)$. Thus, $Apply(A, B, *)$ has asymptotic runtime and memory complexity $O(n)$, leading to an overall asymptotic runtime and memory complexity of $O(n)$ for computing $\otimes_{i=1}^{n} Q_i$. □

Importantly, the terminal values do not need to form a persistent set themselves for the theorem to hold. If they are *contained* in a persistent set, then the sets of all possible

[2] More accurately, B depends on exactly $2n$ variables if it is a matrix QuIDD and n variables if it is a vector QuIDD.

m-element products (i.e. $m \leq n$ for all n-element products in a set Γ) eventually stabilize in the sense that their sizes do not exceed that of Γ. However, this is only true for a fixed m rather than for the sets of products of m elements and fewer.

For QuIDDs A and B, the matrix-matrix and matrix-vector product computations are not as sensitive to terminal values, but depend on the sizes of the QuIDDs. Indeed, the memory and time complexity of this operation is $O((|A||B|)^2)$ [4].

Theorem 3.10 *Consider measuring an n-qubit QuIDD state vector $|\psi\rangle$ using a QuIDD measurement operator M, where both $|\psi\rangle$ and M are constructed via the tensor product of an arbitrary sequence of $O(1)$-sized QuIDD vectors and matrices, respectively. If the terminal node values of the $O(1)$-sized QuIDD vectors or operators are in a persistent set Γ, then the runtime and memory complexity of measuring the QuIDD state vector*[3] *is $O(n^4)$.*

Proof. In Subsection 3.1.6, we showed that runtime and memory complexity for measuring a state vector QuIDD is $O((|A||B|)^2)$, where $|A|$ and $|B|$ are the sizes in nodes of the measurement operator QuIDD and state vector QuIDD, respectively. From Theorem 3.9, the asymptotic memory complexity of both $|A|$ and $|B|$ is $O(n)$, leading to an overall runtime and memory complexity of $O(n^4)$. □

The class of QuIDDs described by Theorem 3.9 and its corollaries, with terminals taken from the set $\{0\} \cup c\mathbb{U}$, encompass a large number of practical quantum state vectors and operators. These include, but are not limited to, any equal superposition of n qubits, any sequence of n qubits in the computational basis states, n-qubit Pauli matrices,

[3] The worst-case bound is rarely reached in practice as demonstrated later.

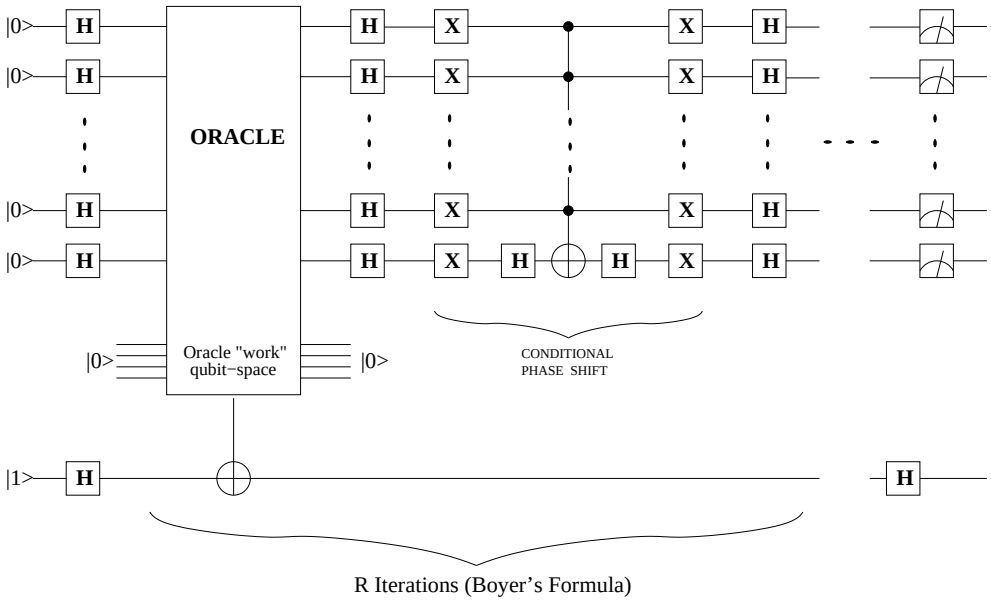


Figure 3.5: Circuit-level implementation of Grover's algorithm

and *n*-qubit Hadamard matrices. The above results suggest a polynomial-sized QuIDD representation of any quantum circuit on *n* qubits in terms of such gates if the number of gates is limited by a constant. In other words, the above sufficient conditions apply if the depth (length) of the circuit is limited by a constant. Our simulation technique may use polynomial memory and runtime in other circumstances as well, as shown in the next subsection.

3.2.2 QuIDD Complexity of Grover's Algorithm

To investigate the power of the QuIDD representation, we used QuIDDPro to simulate Grover's algorithm [33], one of the two major quantum algorithms that have been developed to date. Grover's algorithm searches for a subset of items in an unordered database of *N* items. The only selection criterion available is a black-box predicate or oracle that can be evaluated on any item in the database. The complexity of this evaluation (query)

is unknown, and the overall complexity analysis is performed in terms of queries. In the classical domain, any algorithm for such an unordered search must query the predicate $\Omega(N)$ times. However, Grover's algorithm can perform the search with quantum query complexity $O(\sqrt{N})$, a quadratic improvement. This assumes that a quantum version of the search predicate can be evaluated on a superposition of all database items.

A quantum circuit representation of the algorithm involves five major components: an oracle, a conditional phase shift operator, sets of Hadamard gates, the data qubits, and an oracle qubit. The oracle is a Boolean predicate that acts as a filter, flipping the oracle qubit when it receives as input an *n* bit sequence representing the items being searched for. In quantum circuit form, the oracle is represented as a series of controlled NOT gates with subsets of the data qubits acting as the control qubits and the oracle qubit receiving the action of the NOT gates. Following the oracle circuit, Hadamard gates put the *n* data qubits into an equal superposition of all 2^n items in the database where $2^n = N$. Then a sequence of gates $H^{\otimes n-1}CH^{\otimes n-1}$, where C denotes the conditional phase shift operator, are applied iteratively to the data qubits. Each iteration is termed a *Grover iteration* [51].

Grover's algorithm must be stopped after a particular number of iterations when the probability amplitudes of the states representing the items sought are sufficiently boosted. There must be enough iterations to ensure a successful measurement, but after a certain point the probability of successful measurement starts fading, and later changes periodically. In our experiments, we used the tight bound on the number of iterations formulated by Boyer et al. [14] when the number of solutions M is known in advance: $\lfloor \pi/4\theta \rfloor$ where $\theta = \sqrt{M/N}$. The power of Grover's algorithm lies in the fact that the data qubits store

all $N = 2^n$ items in the database as a superposition, allowing the oracle circuit to "find" all items being searched for *simultaneously*. A circuit implementing Grover's algorithm is shown in Figure 3.5. The algorithm can be summarized as follows, with N denoting the number of elements in the database.

Grover's Algorithm

Step 1. Initialize $n = \lceil \log_2 N \rceil$ qubits to $|0\rangle$ and the *oracle qubit* to $|1\rangle$.

Step 2. Apply the Hadamard transform H to all qubits to put them into a uniform superposition of basis states.

Step 3. Apply the oracle operation which can be implemented as a series of one or more CNOT gates representing the search criteria. The inputs to the oracle circuit feed into the control portions of the CNOT gates, while the oracle qubit is the target qubit for all of the CNOT gates. In this way, if the input to this circuit satisfies the search criteria, the state of the oracle qubit is flipped. For a superposition of inputs, those input basis states that satisfy the search criteria flip the oracle qubit in the composite state-space. The oracle circuit uses ancillary qubits as its workspace, reversibly returning them to their original states (shown as $|0\rangle$ in Fig 3.5). These ancillary qubits will not be operated on by any other step in the algorithm.

Step 4. Apply the H gate to all qubits except the oracle qubit.

Step 5. Apply the conditional phase-shift gate on all qubits except the oracle qubit. This gate negates the probability amplitude of the $|000\ldots0\rangle$ basis state, leaving that of the others unaffected. It can be realized using a combination of X, H and C^{n-1}-NOT gates as shown. A decomposition of the C^{n-1}-NOT into elementary gates is given in [5].

Step 6. Apply the H gate to all gates except the oracle qubit.

Step 7. Repeat steps 3-6 (a single Grover iteration) R times, where $R = \lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \rfloor$ and M is the number of keys matching the search criteria [14].

Step 8. Apply the H gate to the oracle qubit in the last iteration. Measure the first n qubits to obtain the index of the matching key with high probability.

Using explicit vectors and matrices to simulate the above procedure would incur memory and runtime complexities of $\Omega(2^n)$. However, this is not necessarily the case when using QuIDDs. To show this, we present a step-by-step complexity analysis for a QuIDD-based simulation of the procedure.

Steps 1 and 2. Theorem 3.9 implies that the memory and runtime complexity of Step 1 is $O(n)$ because the initial state vector only contains elements in $c\mathbb{U}_k \cup \{0\}$ and is constructed via the tensor product. Step 2 is simply a matrix multiplication of an n-qubit Hadamard matrix with the state vector constructed in Step 1. The Hadamard matrix has memory complexity $O(n)$ by Theorem 3.9. Since the state vector also has memory complexity $O(n)$, further matrix-vector multiplications in Step 2 each have $O(n^4)$ memory and runtime complexity because computing the product of two QuIDDs A and B takes $O((|A||B|)^2)$ time and memory [4]. This upper-bound can be trivially tightened, however. The function of these steps is to put the qubits into an equal superposition. For the n data qubits, this produces a QuIDD with $O(1)$ nodes because an n-qubit state vector representing an equal superposition has only one distinct element, namely $\frac{1}{2^{n/2}}$. Also, applying a Hadamard matrix to the single oracle qubit results in a QuIDD with $O(1)$ nodes because

in the worst-case, the size of a 1-qubit QuIDD is clearly a constant. Since the tensor product is based on the **Apply** algorithm, the result of tensoring the QuIDD representing the data qubits in an equal superposition with the QuIDD for the oracle qubit is a QuIDD containing $O(1)$ nodes.

Steps 3-6. In step 3, the state vector is multiplied by the oracle matrix. Again, the complexity of multiplying two arbitrary QuIDDs A and B is $O((|A||B|)^2)$ [4]. The size of the state vector in Step 3 is $O(1)$. If the size of the oracle is $|A|$, then the memory and runtime complexity of Step 3 is $O(|A|^2)$. Similarly, Steps 4, 5 and 6 will have polynomial memory and runtime complexity in terms of $|A|$ and n.[4] Thus we arrive at the $O(|A|^{16}n^{14})$ worst-case upper-bound for the memory and runtime complexity of the simulation at Step 6. Judging from our empirical data, this bound is typically very loose and pessimistic.

Lemma 3.11 *The memory and runtime complexity of a single Grover iteration in a QuIDD-based simulation is* $O(|A|^{16}n^{14})$.

Proof. Steps $3-6$ make up a single Grover iteration. Since the memory and runtime complexity of a QuIDD-based simulation after completing Step 6 is $O(|A|^{16}n^{14})$, the memory and runtime complexity of a single Grover iteration is $O(|A|^{16}n^{14})$. □

Step 7. This does not involve a quantum operator, but rather it repeats a Grover iteration $R = \lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \rfloor$ times. As a result, Step 7 induces an exponential runtime for the simulation, since the number of Grover iterations is a function of $N = 2^n$. This is acceptable though because an actual quantum computer would also require exponentially many Grover iterations in order to measure one of the matching keys with a high probability

[4] As noted in Step 5, the conditional phase-shift operator can be decomposed into the tensor product of single qubit matrices, giving it memory complexity $O(n)$.

[14]. Ultimately this is the reason why Grover's algorithm only offers a *quadratic* and not an exponential speedup over classical search. Since Lemma 3.11 shows that the memory and runtime complexity of a single Grover iteration is polynomial in the size of the oracle QuIDD, one might guess that the memory complexity of Step 7 is exponential like the runtime. However, it turns out that the size of the state vector does not change from iteration to iteration, as shown below.

Lemma 3.12 *The number of internal nodes of the state vector QuIDD at the end of any Grover iteration i is equal to the number of internal nodes of the state vector QuIDD at the end of Grover iteration $i+1$.*

Proof. Each Grover iteration increases the probability of the states representing matching keys while simultaneously decreasing the probability of the states representing non-matching keys. Therefore, at the end of the first iteration, the state vector QuIDD will have a single terminal node for all the states representing matching keys and one other terminal node, with a lower value, for the states representing non-matching keys (there may be two such terminal nodes for non-matching keys, depending on machine precision). The number of internal nodes of the state vector QuIDD cannot be different at the end of subsequent Grover iterations because a Grover iteration does not change the pattern of probability amplitudes, but only their values. In other words, the same matching states always point to a terminal node whose value becomes closer to 1 after each iteration, while the same non-matching states always point to a terminal node (or nodes) whose value (or values) becomes closer to 0. □

Lemma 3.13 *The total number of nodes in the state vector QuIDD at the end of any Grover iteration i is equal to the total number of nodes in the state vector QuIDD at the end of Grover iteration* $i+1$.

Proof. In proving Lemma 3.12, we showed that the only change in the state vector QuIDD from iteration to iteration is the values in the terminal nodes (not the number of terminal nodes). Therefore, the number of nodes in the state vector QuIDD is always the same at the end of every Grover iteration. □

Corollary 3.14 *In a QuIDD-based simulation, the runtime and memory complexity of any Grover iteration i is equal to the runtime and memory complexity of Grover iteration* $i+1$.

Proof. Each Grover iteration is a series of matrix multiplications between the state vector QuIDD and several operator QuIDDs (Steps $3-6$). The work of Bahar et al. shows that matrix multiplication with ADDs has runtime and memory complexity that is determined solely by the number of nodes in the operands (see Section 3.1.4) [4]. Since the total number of nodes in the state vector QuIDD is always the same at the end of every Grover iteration, the runtime and memory complexity of every Grover iteration is the same. □

Lemmas 3.12 and 3.13 imply that Step 7 does not necessarily induce memory complexity that is exponential in the number of qubits. This important fact is captured in the following theorem.

Theorem 3.15 *The memory complexity of simulating Grover's algorithm using QuIDDs is polynomial in the size of the oracle QuIDD and the number of qubits.*[5]

[5]We do not account for the resources required to construct the QuIDD of the oracle.

Proof. The runtime and memory complexity of a single Grover iteration is $O(|A|^{16}n^{14})$ (Lemma 3.11), which includes the initialization costs of Steps 1 and 2. Also, the structure of the state vector QuIDD does not change from one Grover iteration to the next (Lemmas 3.12 and 3.13). Thus, the overall memory complexity of simulating Grover's algorithm with QuIDDs is $O(|A|^{16}n^{14})$, where $|A|$ is the number of nodes in the oracle QuIDD and n is the number of qubits. □

While any polynomial-time quantum computation can be simulated in polynomial space, the commonly-used linear-algebraic simulation requires $\Omega(2^n)$ space. Also note that the case of an oracle searching for a unique solution (originally considered by Grover) implies that $|A| = n$. Here, most of the searching will be done while constructing the QuIDD of the oracle, which is an entirely classical operation.

As demonstrated experimentally in Section 3.3, for some oracles, simulating Grover's algorithm with QuIDDs has memory complexity $\Theta(n)$. Furthermore, simulation using QuIDDs has worst-case runtime complexity $O(R|A|^{16}n^{14})$, where R is the number of Grover iterations as defined earlier. If $|A|$ grows polynomially with n, this runtime complexity is the same as that of an ideal quantum computer, up to a polynomial factor.

3.3 Empirical Validation

This section discusses problems that arise when implementing a QuIDD-based simulator. It also presents experimental results obtained from actual simulation.

3.3.1 Implementation Issues

Full support of QuIDDs requires the use of complex arithmetic, which can lead to serious problems if numerical precision is not adequately addressed.

Complex Number Arithmetic. At an abstract level, ADDs can support terminals of any numerical type, but CUDD's implementation of ADDs does not. For efficiency reasons, CUDD stores node information in C *union*s which are interpreted numerically for terminals and as child pointers for internal nodes.

However, it is well-known that unions are incompatible with the use of C++ classes because their multiple interpretations hinder the binding of correct destructors. In particular, complex numbers in C++ are implemented as a templated class and are incompatible with CUDD. This was one of the motivations for storing terminal values in an external array.

Numerical Precision. Another important issue is the precision of complex numeric types. Over the course of repeated multiplications, the values of some terminals may become very small and induce round-off errors if the standard IEEE double-precision floating-point types are used. This effect worsens for larger circuits. Unfortunately, such round-off errors can significantly affect the structure of a QuIDD by merging terminals that are only slightly different, or not merging terminals whose values should be equal but differ by a small computational error ε.

The use of approximate comparisons with ε works in certain cases but does not scale well, particularly for creating an equal superposition of states (a standard operation in quantum circuits). In an equal superposition, a circuit with n qubits contains the terminal

Circuit	Hadamards		Conditional	Oracles	
Size n	Initial	Repeated	Phase Shift	1	2
20	80	83	21	99	108
30	120	123	31	149	168
40	160	163	41	199	228
50	200	203	51	249	288
60	240	243	61	299	348
70	280	283	71	349	408
80	320	323	81	399	468
90	360	363	91	449	528
100	400	403	101	499	588

Table 3.1: Size of QuIDDs (no. of nodes) for Grover's algorithm.

value $\frac{1}{2^{n/2}}$ in the state vector. With the IEEE double precision floating-point type, this value will be rounded to 0 at $n = 2048$, preventing the use of epsilons for approximate comparison past $n = 2048$. Furthermore, a static value for epsilon will not work well for different sized circuits. For example, $\varepsilon = 10^{-6}$ may work well for $n = 35$, but not for $n = 40$ because at $n = 40$, all values may be smaller than 10^{-6}. Therefore, to address the problem of precision, QuIDDPro uses an arbitrary precision floating-point type from the GMP library [30] with the C++ complex template. Precision is then limited to the available amount of memory in the system.

3.3.2 Simulating Grover's Algorithm

Before simulation of an instance of Grover's algorithm, we construct the QuIDD representations of Hadamard operators by incrementally tensoring together one-qubit versions of their matrices $n-1$ times to get n-qubit versions. All other QuIDD operators are constructed similarly. Table 3.1 shows all sizes (in nodes) of respective QuIDDs at n-qubits, where $n \in [20..100]$. We observe that memory usage grows linearly in n, and as a result QuIDD-based simulations of Grover's algorithm are not memory-limited even at 100

Oracle 1: Runtime (s)				
n	Oct	MAT	B++	QP
10	80.6	6.64	0.15	0.33
11	2.65e2	22.5	0.48	0.54
12	8.36e2	74.2	1.49	0.83
13	2.75e3	2.55e2	4.70	1.30
14	1.03e4	1.06e3	14.6	2.01
15	4.82e4	6.76e3	44.7	3.09
16	$>$ 24hrs	$>$ 24hrs	1.35e2	4.79
17	$>$ 24hrs	$>$ 24hrs	4.09e2	7.36
18	$>$ 24hrs	$>$ 24hrs	1.23e3	11.3
19	$>$ 24hrs	$>$ 24hrs	3.67e3	17.1
20	$>$ 24hrs	$>$ 24hrs	1.09e4	26.2
21	$>$ 24hrs	$>$ 24hrs	3.26e4	39.7
22	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	60.5
23	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	92.7
24	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	1.40e2
25	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	2.08e2
26	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	3.12e2
27	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	4.72e2
28	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	7.07e2
29	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	1.08e3
30	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	1.57e3
31	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	2.35e3
32	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	3.53e3
33	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	5.23e3
34	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	7.90e3
35	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	1.15e4
36	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	1.71e4
37	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	2.57e4
38	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	3.82e4
39	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	5.64e4
40	$>$ 24hrs	$>$ 24hrs	$>$ 24hrs	8.23e4

(a)

Oracle 1: Peak Memory Usage (MB)				
n	Oct	MAT	B++	QP
10	2.64e-2	1.05e-2	3.52e-2	9.38e-2
11	5.47e-2	2.07e-2	8.20e-2	0.121
12	0.105	4.12e-2	0.176	0.137
13	0.213	8.22e-2	0.309	0.137
14	0.426	0.164	0.559	0.137
15	0.837	0.328	1.06	0.137
16	1.74	0.656	2.06	0.145
17	3.34	1.31	4.06	0.172
18	4.59	2.62	8.06	0.172
19	13.4	5.24	16.1	0.172
20	27.8	10.5	32.1	0.172
21	55.6	NA	64.1	0.195
22	NA	NA	1.28e2	0.207
23	NA	NA	2.56e2	0.207
24	NA	NA	5.12e2	0.223
25	NA	NA	1.02e3	0.230
26	NA	NA	$>$ 1.5GB	0.238
27	NA	NA	$>$ 1.5GB	0.254
28	NA	NA	$>$ 1.5GB	0.262
29	NA	NA	$>$ 1.5GB	0.277
30	NA	NA	$>$ 1.5GB	0.297
31	NA	NA	$>$ 1.5GB	0.301
32	NA	NA	$>$ 1.5GB	0.305
33	NA	NA	$>$ 1.5GB	0.320
34	NA	NA	$>$ 1.5GB	0.324
35	NA	NA	$>$ 1.5GB	0.348
36	NA	NA	$>$ 1.5GB	0.352
37	NA	NA	$>$ 1.5GB	0.371
38	NA	NA	$>$ 1.5GB	0.375
39	NA	NA	$>$ 1.5GB	0.395
40	NA	NA	$>$ 1.5GB	0.398

(b)

Table 3.2: Simulating Grover's algorithm with n qubits using Octave (Oct), MATLAB (MAT), Blitz++ (B++) and our simulator QuIDDPro (QP). $>$ 24hrs indicates that the runtime exceeded our cutoff of 24 hours. $>$ 1.5GB indicates that the memory usage exceeded our cutoff of 1.5GB. Simulation runs that exceed the memory cutoff can also exceed the time cutoff, though we give memory cutoff precedence. NA indicates that after a cutoff of one week, the memory usage was still steadily growing, preventing a peak memory usage measurement.

Oracle 2: Runtime (s)				
n	Oct	MAT	B++	QP
13	1.39e3	1.31e2	2.47	0.617
14	3.75e3	7.26e2	5.42	0.662
15	1.11e4	4.27e3	11.7	0.705
16	3.70e4	2.23e4	24.9	0.756
17	$>$ 24hrs	$>$ 24hrs	53.4	0.805
18	$>$ 24hrs	$>$ 24hrs	1.13e2	0.863
19	$>$ 24hrs	$>$ 24hrs	2.39e2	0.910
20	$>$ 24hrs	$>$ 24hrs	5.15e2	0.965
21	$>$ 24hrs	$>$ 24hrs	1.14e3	1.03
22	$>$ 24hrs	$>$ 24hrs	2.25e3	1.09
23	$>$ 24hrs	$>$ 24hrs	5.21e3	1.15
24	$>$ 24hrs	$>$ 24hrs	1.02e4	1.21
25	$>$ 24hrs	$>$ 24hrs	2.19e4	1.28
26	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.35
27	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.41
28	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.49
29	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.55
30	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.63
31	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.71
32	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.78
33	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.86
34	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	1.94
35	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.03
36	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.12
37	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.21
38	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.29
39	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.37
40	$>$ 24hrs	$>$ 24hrs	$>$ 1.5GB	2.47

(a)

Oracle 2: Peak Memory Usage (MB)				
n	Oct	MAT	B++	QP
13	0.218	8.22e-2	0.252	0.137
14	0.436	0.164	0.563	0.141
15	0.873	0.328	1.06	0.145
16	1.74	0.656	2.06	0.172
17	3.34	1.31	4.06	0.176
18	4.59	2.62	8.06	0.180
19	13.4	5.24	16.1	0.180
20	27.8	10.5	32.1	0.195
21	55.6	NA	64.1	0.199
22	NA	NA	1.28e2	0.207
23	NA	NA	2.56e2	0.215
24	NA	NA	5.12e2	0.227
25	NA	NA	1.02e3	0.238
26	NA	NA	$>$ 1.5GB	0.246
27	NA	NA	$>$ 1.5GB	0.256
28	NA	NA	$>$ 1.5GB	0.266
29	NA	NA	$>$ 1.5GB	0.297
30	NA	NA	$>$ 1.5GB	0.301
31	NA	NA	$>$ 1.5GB	0.305
32	NA	NA	$>$ 1.5GB	0.324
33	NA	NA	$>$ 1.5GB	0.328
34	NA	NA	$>$ 1.5GB	0.348
35	NA	NA	$>$ 1.5GB	0.352
36	NA	NA	$>$ 1.5GB	0.375
37	NA	NA	$>$ 1.5GB	0.375
38	NA	NA	$>$ 1.5GB	0.395
39	NA	NA	$>$ 1.5GB	0.398
40	NA	NA	$>$ 1.5GB	0.408

(b)

Table 3.3: Simulating Grover's algorithm with n qubits using Octave (Oct), MATLAB (MAT), Blitz++ (B++) and our simulator QuIDDPro (QP). $>$ 24hrs indicates that the runtime exceeded our cutoff of 24 hours. $>$ 1.5GB indicates that the memory usage exceeded our cutoff of 1.5GB. Simulation runs that exceed the memory cutoff can also exceed the time cutoff, though we give memory cutoff precedence. NA indicates that after a cutoff of one week, the memory usage was still steadily growing, preventing a peak memory usage measurement.

qubits. Note that this is consistent with Theorem 3.9.

With the operators constructed, simulation can proceed. Tables 3.2a and 3.2b show performance measurements for simulating Grover's algorithm with an oracle circuit that

searches for one item out of 2^n. QuIDDPro achieves asymptotic memory savings compared to qubit-wise implementations (see Section 2.1) of Grover's algorithm using Blitz++, a high-performance numerical linear algebra library for C++ [75], MATLAB, and Octave, a mathematical package similar to MATLAB. The overall runtimes are still exponential in n because Grover's algorithm entails an exponential number of iterations, even on an actual quantum computer [14]. We also studied a "mod-1024" oracle circuit that searches for elements whose ten least significant bits are 1 (see Tables 3.3a and 3.3b). Results were produced on a 1.2GHz AMD Athlon with 1GB RAM running Linux. Memory usage for MATLAB and Octave is lower-bounded by the size of the state vector and conditional phase shift operator; Blitz++ and QuIDDPro memory usage is measured as the size of the entire program. Simulations using MATLAB and Octave past 15 qubits timed out at 24 hours.

3.3.3 Impact of Grover Iterations

To verify that the QuIDDPro simulation resulted in the exact number of Grover iterations required to generate the highest probability of measuring the items being sought as per the Boyer et al. formulation [14], we tracked the probabilities of these items as a function of the number of iterations. For this experiment, we used four different oracle circuits, each with 11,12, and 13 qubit circuits. The first oracle is called "Oracle N" and represents an oracle in which all the data qubits act as controls to flip the oracle qubit (this oracle is equivalent to Oracle 1 in the last subsection). The other oracle circuits are "Oracle N-1", "Oracle N-2", and "Oracle N-3", which all have the same structure as Oracle N minus 1,2, and 3 controls, respectively. As described earlier, each removal of a control doubles the

number of items being searched for in the database. For example, Oracle N-2 searches for 4 items in the data set because it recognizes the bit pattern 111...1dd.

Oracle	11 Qubits	12 Qubits	13 Qubits
N	25	35	50
$N-1$	17	25	35
$N-2$	12	17	25
$N-3$	8	12	17

Table 3.4: Number of Grover iterations at which Boyer et al. [14] predict the highest probability of measuring one of the items sought.

Table 3.4 shows the optimal number of iterations produced with the Boyer et al. formulation for all the instances tested. Figure 3.6 plots the probability of successfully finding any of the items sought against the number of Grover iterations. In the case of Oracle N, we plot the probability of measuring the single item being searched for. Similarly, for oracles N-1, N-2, and N-3, we plot the probability of measuring any one of the 2, 4, and 8 items being searched for, respectively. By comparing the results in Table 3.4 with those in Figure 3.6, it can be easily verified that QuIDDPro uses the correct number of iterations at which measurement is most likely to produce items sought. Also notice that the probabilities, as a function of the number of iterations, follow a sinusoidal curve. It is therefore important to terminate at the exact optimal number of iterations not only from an efficiency standpoint but also to prevent the probability amplitudes of the items being sought from lowering back down toward 0.

3.4 Summary

We proposed and tested a new technique for simulating quantum circuits using a data structure called a QuIDD. We have shown that QuIDDs enable practical, generic and rea-

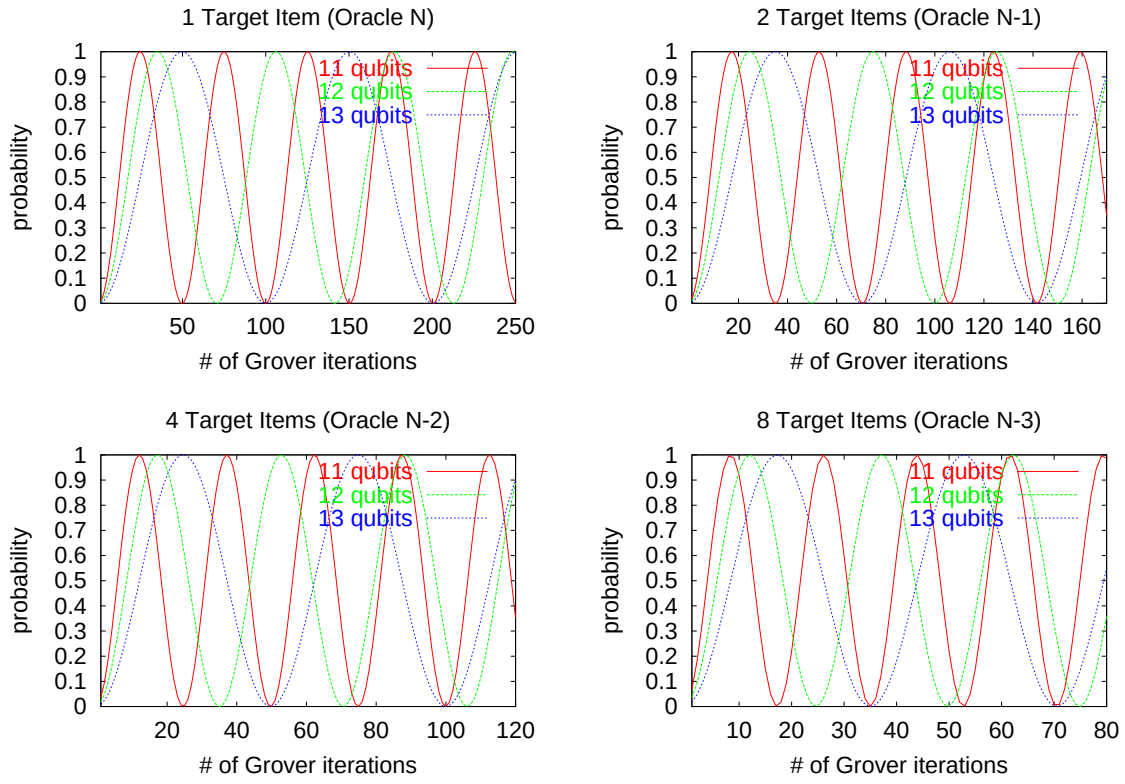


Figure 3.6: Probability of successful search for one, two, four and eight items as a function of the number of iterations after which the measurement is performed (11, 12 and 13 qubits). Note that the minima and maxima of the empirical sine curves match the predictions in Table 3.4.

sonably efficient simulation of quantum computation. Their key advantages are faster execution and lower memory usage. In our experiments, QuIDDPro achieves exponential memory savings compared to other known techniques.

This result explores the limitations of quantum computing, and we have subsequently expanded this investigation in [77]. Classical computers have the advantage that they are not subject to quantum measurement and errors. Thus, when competing with quantum computers, classical computers can simply run ideal error-free quantum algorithms (as was done in Section 3.3), allowing techniques such as QuIDDs to exploit the symmetries found

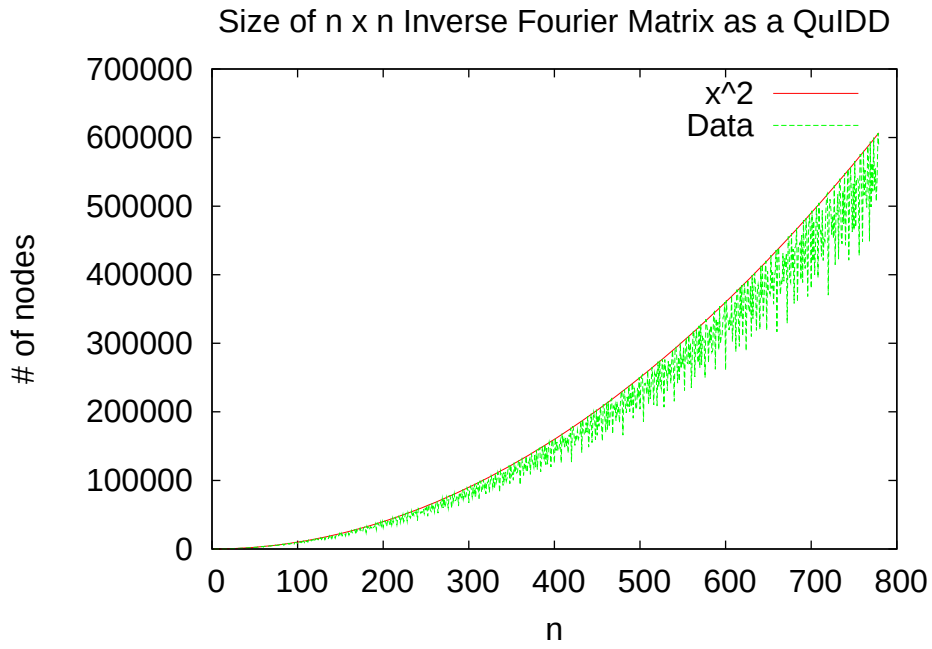


Figure 3.7: Growth of inverse Quantum Fourier Transform matrix in QuIDD form. $N = 2^n$ for n qubits.

in ideal quantum computation. On the other hand, quantum computation still has certain operators which cannot be represented using only polynomial resources on a classical computer, even with QuIDDs. Examples of such operators include the quantum Fourier transform (QFT) and its inverse which are used in Shor's number factoring algorithm [65]. Figure 3.7 shows the growth in number of nodes of the N by N inverse QFT as a QuIDD. Since $N = 2^n$ where n is the number of qubits, this QuIDD exhibits exponential growth with a linear increase in qubits. Therefore, the inverse QFT will cause QuIDDPro to have exponential runtime and memory requirements when simulating Shor's algorithm.

Another challenging aspect of quantum simulation is the impact of errors due to defects in circuit components, and environmental effects such as decoherence. Error simulation appears to be essential for modeling actual quantum computational devices. It may, how-

ever, prove to be difficult since errors can alter the symmetries exploited by QuIDDs. An important step in studying errors is to extend QuIDDs to encompass the density matrix representation, and this extension is described in the next chapter.

CHAPTER IV

Density Matrix Simulation with QuIDDs

This chapter extends QuIDD-based quantum circuit simulation to the density matrix representation and is based on the work published in [76, 78]. As noted earlier (Subsection 1.2.1), the density matrix representation is crucial in capturing interactions between quantum states and the environment, such as noise. In addition to the standard set of operations required to simulate with the state-vector model, including matrix multiplication and the tensor product, simulation with the density matrix model requires the outer product and the partial trace. The outer product is used in the initialization of qubit density matrices, while the partial trace allows a simulator to differentiate qubit states coupled to noisy environments or other unwanted states. The partial trace is invaluable in error modeling since it facilitates descriptions of single qubit states that have been affected by noise and other phenomena [51]. As a result, we derive algorithms to implement the outer product and the partial trace using QuIDDs.

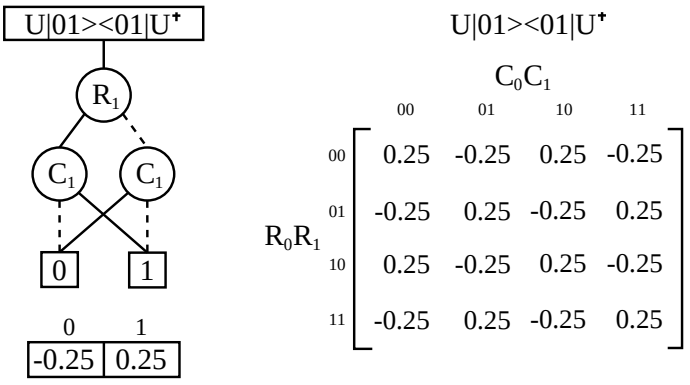
We also describe a set of quantum circuit benchmarks that incorporate errors, error correction, reversible logic, quantum communication, and quantum search. To empirically evaluate the improvements offered by QuIDD-based density matrix simulation, we use

these benchmarks to compare QuIDDPro with an array-based density matrix simulator called QCSim [12] that makes use of qubit-wise multiplication algorithms. Performance data from both simulators show that our new graph-based algorithms far outperform the array-based approach for the given benchmarks. It should be noted, however, that not all quantum circuits can be simulated efficiently with QuIDDs. A useful class of matrices and vectors which can be manipulated efficiently by QuIDDs was formally described in the previous section and is restated below. For some matrices and vectors outside of this class, QuIDD-based simulation can be up to three times slower due to the overhead of following pointers in the QuIDD data structure.

4.1 Existing QuIDD Properties and Density Matrices

Although the density matrix representation can be invaluable for simulating environmental noise in quantum circuits, like the state vector representation, it is plagued by runtime and memory complexity that grows exponentially with the number of qubits in the worst case. As discussed in Subsection 1.2.1, a straightforward linear-algebraic simulation using density matrices requires $O(2^{2n})$ time and memory resources. Since QuIDDs have proven useful in reducing this complexity in the state vector paradigm, it is only natural to extend QuIDDs to the density matrix model in an attempt to reduce the simulation complexity of this important model in practical cases. Before proceeding to the new extensions, it is instructive to first review what is already in place that can be re-used in the density matrix representation.

Figure 4.1a shows the QuIDD that results from applying $U = H \otimes H$ to an outer product as $U|01\rangle\langle 01|U^{\dagger}$. The R_i nodes of the QuIDD encode the binary indices of the rows in the



(a) (b)

Figure 4.1: (a) QuIDD for the density matrix resulting from $U|01\rangle\langle 01|U^{\dagger}$, where $U = H \otimes H$, and (b) its explicit matrix form.

explicit matrix. Similarly, the C_i nodes encode the binary indices of the columns. Solid lines leaving a node denote the positive cofactor of the index bit variable (a value of 1), while dashed lines denote the negative cofactor (a value of 0). Terminal nodes correspond to the value of the element in the explicit matrix whose binary row/column indices are encoded by the path that was traversed.

Notice that the first and second pairs of rows of the explicit matrix in Figure 4.1b are the same, as are the first and second pairs of columns. This redundancy is captured by the QuIDD in Figure 4.1a because the QuIDD does not contain any R_0 or C_0 nodes. In other words, the values and their locations in the explicit matrix can be completely determined without the superfluous knowledge of the first row and column index bits.

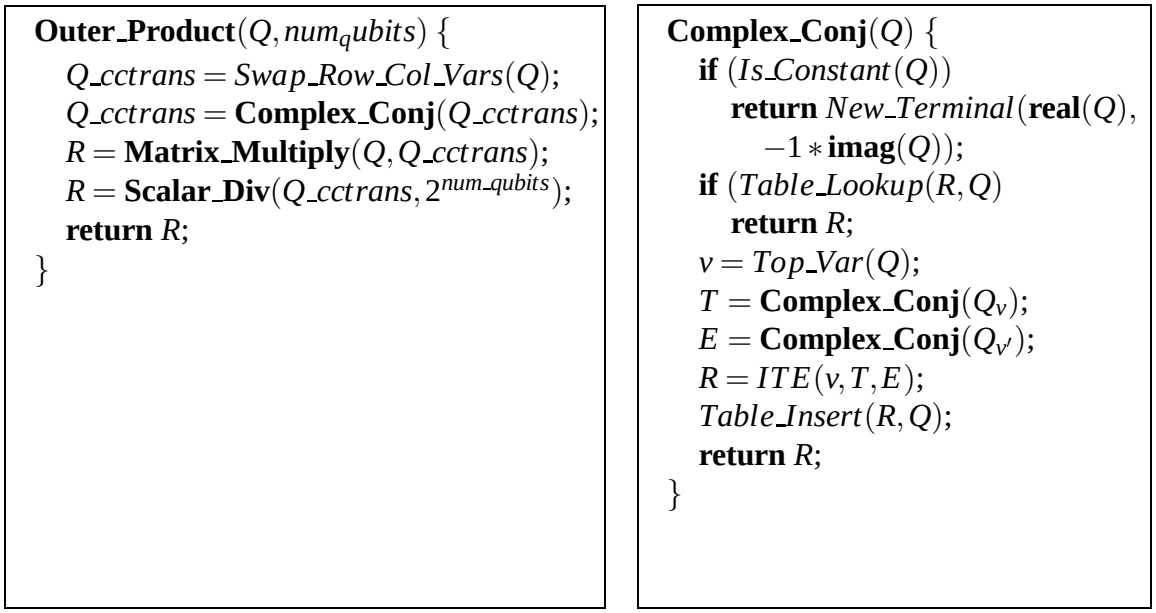
Measurement, matrix multiplication, addition, scalar products, the tensor product, and other operations involving QuIDDs are variations of the well-known **Apply** algorithm (Chapter III). Vectors and matrices with large blocks of repeated values can be manipulated in QuIDD form quite efficiently with these operations. Section 3.2 provides a formal

description of a class of vectors and matrices that is simulated efficiently with QuIDDs. Since QuIDDs already have the capability to represent matrices and multiply them, extending QuIDDs to encompass the density matrix representation requires algorithms for the outer product and the partial trace.

4.2 QuIDD-based Outer Product

The outer product involves matrix multiplication between a column vector and its complex-conjugate transpose. Since a column vector QuIDD only depends on row variables, the transpose can be accomplished by swapping the row variables with column variables. The complex conjugate can then be performed with a DFS traversal that replaces terminal node values with their complex conjugates. The original column vector QuIDD is then multiplied by its complex-conjugate transpose using the matrix multiply operation previously defined for QuIDDs (Subsection 3.1.4). Pseudo-code for this algorithm is given in Figure 4.2. Notice that before the result is returned, it is divided by 2^{num_qubits}, where *num_qubits* is the number of qubits represented by the QuIDD vector. This is done because a QuIDD that only depends on n row variables can be viewed as either a $2^n \times 1$ column vector or a $2^n \times 2^n$ matrix in which all columns are the same. Since matrix multiplication is performed in terms of the latter case [80, 82, 4], the result of the outer product contains values that are multiplied by an extra factor of 2^n, which must be normalized.

Although QuIDDs enable efficient simulation for a class of matrices and vectors in the state-vector paradigm, it must be shown that the corresponding density matrix version of this class can also be simulated efficiently. Since state-vectors are converted to density matrices via the outer product, this requires proving that the outer product of a QuIDD



(a) (b)

Figure 4.2: Pseudo-code for (a) the QuIDD outer product and (b) its complex conjugation helper function **Complex_Conj**. The code for **Scalar_Div** is the same as **Complex_Conj**, except that in the terminal node case it returns the value of the terminal divided by a scalar. Other functions are typical ADD operations [4, 66].

vector in this class with its complex-conjugate transpose results in a QuIDD density matrix of size polynomial in the number of qubits.

Theorem 4.1 *Given an n-qubit QuIDD state-vector created from tensor products of QuIDDs with $O(1)$ nodes whose terminal values are in a persistent set, the outer product of this QuIDD with its complex-conjugate transpose produces a QuIDD matrix with polynomially many nodes in n.*

Proof. Since the given QuIDD state-vector's terminal values are in a persistent set, the number of nodes in the QuIDD is $O(n)$ (Theorem 3.9). Consider the pseudo-code for the QuIDD outer product shown in Figure 4.2a. The first operation is to create a transposed copy of the QuIDD state-vector. Transposition only requires remapping the internal vari-

able nodes to represent column variables instead of row variables. This can be done in one pass over all the nodes in the QuIDD state-vector (Subsection 3.1.5). Since the number of nodes is $O(n)$, this operation has $O(n)$ runtime complexity and creates a transposed copy with $O(n)$ nodes. The next operation is to complex-conjugate the transposed QuIDD copy. As evidenced by the pseudo-code for complex conjugation of QuIDDs in Figure 4.2b, this involves a single recursive pass over all nodes. All internal nodes are returned unchanged with the $O(1)$ ADD ITE operation [4], whereas the complex-conjugates of the terminals are returned when they are reached. Since the number of nodes in the transposed QuIDD copy is $O(n)$, the runtime complexity of this operation is $O(n)$ and results in a new QuIDD with $O(n)$ nodes. Next, QuIDD matrix multiplication is performed on the QuIDD state-vector and its complex-conjugate transpose to produce the QuIDD density matrix. It has been proven that QuIDD matrix multiplication of some QuIDD A with $|A|$ nodes and another QuIDD B with $|B|$ nodes has runtime complexity $O((|A||B|)^2)$ and results in a QuIDD with $O((|A||B|)^2)$ nodes (Subsection 3.1.4). Since the QuIDD state-vector and its complex-conjugate transpose each have $O(n)$ nodes, the matrix multiplication step has runtime complexity $O(n^4)$. The final normalization step of the outer product is a scalar division of the terminal values. Like QuIDD complex conjugation, this operation is implemented by a single recursive pass over the QuIDD, but when the terminals are reached the scalar division result is returned. Since the QuIDD density matrix has $O(n^4)$ nodes, this operation has runtime complexity $O(n^4)$. Based on the complexity of all steps in the QuIDD outer product algorithm, the overall runtime complexity of the QuIDD outer product is $O(n^4)$ and results in a QuIDD density matrix with $O(n^4)$ nodes. □

4.3 QuIDD-based Partial Trace

To motivate the QuIDD-based partial trace algorithm, we note how the partial trace can be performed with explicit matrices. The trace of a matrix A is the sum of A's diagonal elements. To perform the partial trace over a particular qubit in an n-qubit density matrix, the trace operation can be applied iteratively to sub-matrices of the density matrix. Each sub-matrix is composed of four elements with row indices $r0s$ and $r1s$, and column indices $c0d$ and $c1d$, where r, s, c, and d are arbitrary sequences of bits which index the n-qubit density matrix.

Tracing over these sub-matrices has the effect of reducing the dimensionality of the density matrix by one qubit. A well-known ADD operation which reduces the dimensionality of a matrix is the **Abstract** operation [4]. Given an arbitrary ADD f, abstraction of variable x_i eliminates all internal nodes of f which represent x_i by combining the positive and negative cofactors of f with respect to x_i using some binary operation. In other words, $\textbf{Abstract}(f, x_i, op) = f_{x_i}\ op\ f_{x_i'}$.

For QuIDDs, there is a one-to-one correspondence between a qubit on wire i (wires are labeled top-down starting at 0) and variables R_i and C_i. So at first glance, one may suspect that the partial trace of qubit i in f can be achieved by performing $\textbf{Abstract}(f, R_i, +)$ followed by $\textbf{Abstract}(f, C_i, +)$. However, this will add the rows determined by qubit i independently of the columns. The desired behavior is to perform the diagonal addition of sub-matrices by accounting for both the row and column variables due to *i simultaneously*. The pseudo-code to perform the partial trace correctly is depicted in Figure 4.3. In comparing this with the pseudo-code for the **Abstract** algorithm [4], the main differ-

```
Ptrace(Q, qubit_index) {
  if(Is_Constant(Q))
    return Q;
  top_q = Top_Var
  if (qubit_index < Index(top_q)) {
    R = Apply(Q, Q, +);
    return R;
  }
  if (Table_Lookup(R, Q, qubit_index)
    return R;
  T = Q_top_q;
  E = Q_top_q';
  if (qubit_index == Index(top_q)) {
    if (Is_Constant(T) or Index(T) > Index(Q)+1)
      r1 = T;
    else {
      top_T = Top_Var(T);
      r1 = T_top_T;
    }
    if (Is_Constant(E) or Index(E) > Index(Q)+1)
      r2 = E;
    else {
      top_E = Top_Var(E);
      r2 = E_top_E';
    }
    R = Apply(r1, r2, +);
    Table_Insert(R, Q, qubit_index);
    return R;
  }
  else {   /* (qubit_index > Index(top_q)) */
    r1 = Ptrace(T, qubit_index);
    r2 = Ptrace(E, qubit_index);
    R = ITE(top_q, r1, r2);
    Table_Insert(R, Q, qubit_index);
    return R;
  }
}
```

Figure 4.3: Pseudo-code for the QuIDD partial trace. The index of the qubit being traced-over is *qubit_index*.

ence is that when R_i corresponding to qubit i is reached, we take the positive and negative cofactors *twice* before making the recursive call. Since the interleaved variable ordering of QuIDDs guarantees that C_i immediately follows R_i [80, 82], taking the positive and negative cofactors twice simultaneously abstracts both the row and column variables for qubit i, achieving the desired result of summing diagonals. In other words, for a QuIDD f, the partial trace over qubit i is **Ptrace**$(f,i) = f_{R_iC_i} + f_{R_i'C_i'}$. Note that in the pseudo-code there are checks for the special case when no internal nodes in the QuIDD represent C_i. Not shown in the pseudo-code is book-keeping which shifts up the variables in the resulting QuIDD to fill the hole in the ordering left by the row and column variables that were traced-over.

As in the case of the outer product, the QuIDD partial trace algorithm has efficient runtime and memory complexity in the size of the QuIDD being traced-over, which we now show.

Theorem 4.2 *Given an n-qubit density matrix QuIDD A with $|A|$ nodes, any qubit represented in the matrix can be traced-over with runtime complexity $O(|A|)$ and results in a density matrix QuIDD with $O(|A|)$ nodes.*

Proof. Consider the pseudo-code for the QuIDD partial trace algorithm in Figure 4.3. The algorithm performs a recursive traversal over the nodes in the QuIDD density matrix and takes certain actions when special cases are encountered. If a node is encountered which corresponds to a qubit preceded by the traced-over qubit in the variable ordering,[1] then recursion stops and the sub-graph is added to itself with the ADD **Apply** algorithm [17].

[1]Recall that there is a one-to-one correspondence between a qubit on wire i and variables R_i and C_i

This operation has runtime complexity $O(|A|)$ and results in a new sub-graph with $O(|A|)$ nodes. Next, if the partial trace of the current sub-graph has already been computed, then recursion stops and the pre-computed result is simply looked up in the computed table cache and returned. This operation has runtime complexity $O(1)$ and returns a sub-graph with $O(|A|)$ nodes [17]. If there is no entry in the computed table cache, the algorithm checks if the current node's variable corresponds to the qubit to be traced-over. If so, **Apply** is used to add the node's children or children's children, which again has $O(|A|)$ runtime and memory complexity. If the current node does not correspond to the qubit being traced-over, then the partial trace algorithm is called recursively on the node's children. Since all the other special cases stop recursion and involve an **Apply** operation, then the overall runtime complexity of the partial trace algorithm is $O(|A|)$ and results in a new QuIDD with $O(|A|)$ nodes. □

4.4 Experimental Results

We consider a number of quantum circuit benchmarks which cover errors, error correction, reversible logic, communication, and quantum search. We devised some of these benchmarks, while others are drawn from NIST [12] and from a site devoted to reversible circuits [49]. For every benchmark, the simulation performance of QuIDDPro is compared with NIST's QCSim quantum circuit simulator, which utilizes an explicit array-based computational engine. The results indicate that QuIDDPro far outperforms QCSim. All experiments are performed on a 1.2GHz AMD Athlon workstation with 1GB of RAM running Linux.

4.4.1 Reversible Circuits

Here we examine the performance of QuIDDPro simulating a set of reversible circuits, which are quantum circuits that perform classical operations [51]. Specifically, if the input qubits of a quantum circuit are all in the computational basis (i.e. they have only $|0\rangle$ or $|1\rangle$ values), there is no quantum noise, and all the gates are "k-CNOT gates" with $k = 0$ for X, $k = 1$ for CNOT, etc. [63], then the output qubits and all intermediate states will also be in the computational basis. Such a circuit results in a classical logic operation which is reversible in the sense that the inputs can always be derived from the outputs and the circuit function. Reversibility comes from the fact that all quantum operators must be unitary and therefore all have inverses [51].

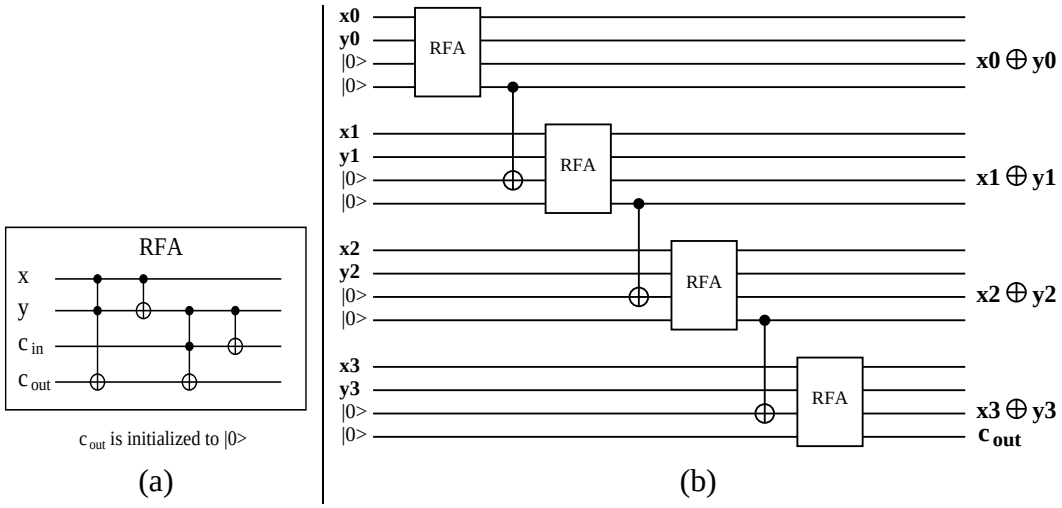


Figure 4.4: (a) An implementation of a reversible full-adder (RFA), and (b) a reversible 4-bit ripple-carry adder which uses the RFA as a module. The reversible ripple-carry adder circuit computes the binary sum of two 4-bit numbers: $x_3x_2x_1x_0 \oplus y_3y_2y_1y_0$. c_{out} is the final carry bit output from the addition of the most-significant bits (x_3 and y_3).

The first reversible benchmark we consider is a reversible 4-bit ripple-carry adder which is depicted in Figure 4.4. Since the size of a QuIDD is sensitive to the arrange-

ment of different values of matrix elements, we simulate the adder with varied input values ("rc_adder1" through "rc_adder4"). This is also done for other benchmarks. Two other reversible benchmarks we simulate contain fewer qubits but more gates than the ripple-carry adder. One of these benchmarks is a 12-qubit reversible circuit that outputs a $|1\rangle$ on the last qubit if and only if the number of $|1\rangle$'s in the input qubits is 3, 4, 5, or 6 ("9sym1" through "9sym5") [49]. The other benchmark is a 15-qubit reversible circuit that generates the classical Hamming code of the input qubits ("ham15_1" through "ham15_3") [49].

Performance results for all of these benchmarks are shown in Table 4.1. QuIDDPro significantly outperforms QCSim in every case. In fact for circuits of 14 or more qubits, QCSim requires more than 2GB of memory. Since QCSim uses an explicit array-based engine, it is insensitive to the arrangement and values of elements in matrices. Therefore, one can expect QCSim to use more than 2GB of memory for *any* benchmark with 14 or more qubits, regardless of the circuit functionality and input values. Another interesting result is that even though QuIDDPro is, in general, sensitive to the arrangement and values of matrix elements, the data indicate that QuIDDPro is insensitive to varied inputs on the same circuit for error-free reversible benchmarks. However, QuIDDPro still compresses the tremendous amount of redundancy present in these benchmarks.

4.4.2 Error Correction and Communication

Now we analyze the performance of QuIDDPro on simulations that incorporate errors and error correction. We consider some simple benchmarks that encode single qubits into Steane's 7-qubit error-correcting code [69] and some more complex benchmarks that use the Steane code to correct a combination of bit-flip and phase-flip errors in a half-adder

Benchmark	No. of Qubits	No. of Gates	QuIDDPro		QCSim	
			Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
rc_adder1	16	24	0.44	0.0625	—	$> 2GB$
rc_adder2	16	24	0.44	0.0625	—	$> 2GB$
rc_adder3	16	24	0.44	0.0625	—	$> 2GB$
rc_adder4	16	24	0.44	0.0625	—	$> 2GB$
9sym1	12	29	0.2	0.0586	8.01	128.1
9sym2	12	29	0.2	0.0586	8.02	128.1
9sym3	12	29	0.2	0.0586	8.04	128.1
9sym4	12	29	0.2	0.0586	8	128.1
9sym5	12	29	0.2	0.0586	7.95	128.1
ham15_1	15	148	1.99	0.121	—	$> 2GB$
ham15_2	15	148	2.01	0.121	—	$> 2GB$
ham15_3	15	148	1.99	0.121	—	$> 2GB$

Table 4.1: Performance results for QuIDDPro and QCSim on the reversible circuit benchmarks. $> 2GB$ indicates that a memory usage cutoff of 2GB was exceeded.

and Grover's quantum search algorithm [33]. Secure quantum communication is also considered here because eavesdropping disrupts a quantum channel and can be treated as an error.

The first two benchmarks "steaneX" and "steaneZ" encode a single logical qubit as seven physical qubits with the Steane code and simulate the effect of a probabilistic bit-flip and phase-flip error, respectively [12]. "steaneZ" contains 13 qubits which are initialized to the state $0.866025|0000000000000\rangle + 0.5|0000001000000\rangle$. A combination of gates apply a probabilistic phase-flip on one of the qubits and calculate the error syndrome and error rate. "steaneX" is a 12-qubit version of the same circuit that simulates a probabilistic bit-flip error.

A more complex benchmark that we simulate is a reversible half-adder with three logical qubits that are encoded into 21 physical qubits with the Steane code. Additionally, three ancillary qubits are used to track the error rate, giving a total circuit size of 24 qubits. "hadder1_bf1" through "hadder3_bf3" simulate the half-adder with different

numbers of bit-flip errors on various physical qubits in the encoding of one of the logical qubit inputs. Similarly, “hadder1_pf1” through “hadder3_pf3” simulate the half-adder with various phase-flip errors.

Another large benchmark used is an instance of Grover’s quantum search algorithm [33]. Whereas the simulations of this algorithm described in the last section utilized the state vector representation, this experiment utilizes the density matrix representation. The oracle in this benchmark searches for one element in a database of four items. It has two logical data qubits and one logical, ancillary oracle-qubit which are all encoded with the Steane code. Like the half-adder circuit, this results in a total circuit size of 24 qubits. “grover_s1” simulates the circuit with the encoded qubits in the absence of errors. “grover_s_bf1” and “grover_s_pf1” introduce and correct a bit-flip and phase-flip error, respectively, on one of the physical qubits in the encoding of the logical oracle qubit.

In addition to error modeling and error correction for computational circuits, another important application is secure communication using quantum cryptography. The basic concept is to use entanglement to distribute a shared key. Eavesdropping constitutes a measurement of the quantum state representing the key, disrupting the quantum state. This disruption can be detected by the legitimate communicating parties. Since actual implementations of quantum key distribution have already been demonstrated [25], efficient simulation of these protocols may play a key role in exploring possible improvements. Therefore, we present two benchmarks which implement BB84, one of the earliest quantum key distribution protocols [6]. “bb84Eve” accounts for the case in which an eavesdropper is present (see Figure 4.5) and contains 9 qubits, whereas “bb84NoEve” accounts

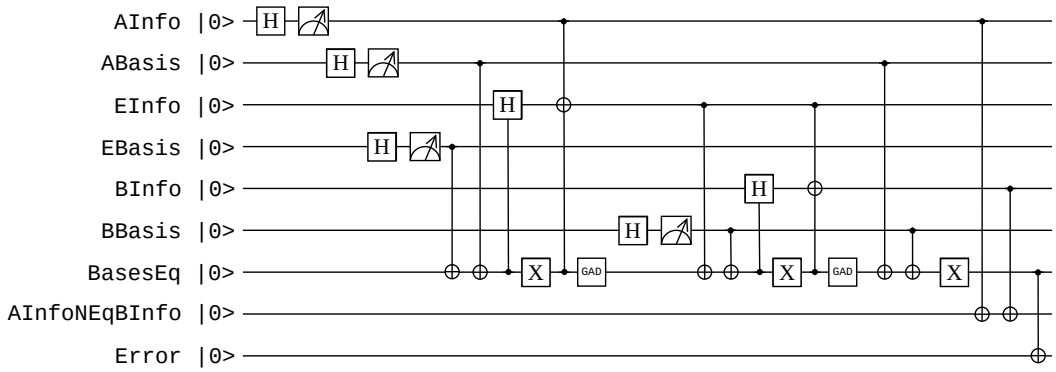


Figure 4.5: Quantum circuit for the “bb84Eve” benchmark.

for the case in which no eavesdropper is present and contains 7 qubits. In both circuits, all qubits are traced-over at the end except for two qubits reserved to track whether or not the legitimate communicating parties successfully shared a key (BasesEq) and the error due to eavesdropping (Error).

Performance results for all of these benchmarks are shown in Table 4.2. Again, QuIDD-Pro significantly outperforms QCSim on all benchmarks except for “bb84Eve” and “bb84NoEve.” The performance of QuIDDPro and QCSim is about the same for these benchmarks. The reason is that these benchmarks contain fewer qubits than all of the others. Since each additional qubit doubles the size of an explicit density matrix, QCSim has difficulty simulating the larger Steane encoded benchmarks.

4.4.3 Scalability and Quantum Search

To analyze scalability with the number of input qubits, we turn to quantum circuits containing a variable number of input qubits. In particular, we reconsider Grover’s quantum search algorithm. However, for these instances of quantum search, the qubits are not encoded with the Steane code, and errors are not introduced. The oracle performs the same function as the one described in the last subsection except that the number of data qubits

Benchmark	No. of Qubits	No. of Gates	QuIDDPro		QCSim	
			Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
steaneZ	13	143	0.6	0.672	287	512
steaneX	12	120	0.27	0.68	53.2	128
hadder_bf1	24	49	18.3	1.48	—	$> 2GB$
hadder_bf2	24	49	18.7	1.48	—	$> 2GB$
hadder_bf3	24	49	18.7	1.48	—	$> 2GB$
hadder_pf1	24	51	21.2	1.50	—	$> 2GB$
hadder_pf2	24	51	21.2	1.50	—	$> 2GB$
hadder_pf3	24	51	20.7	1.50	—	$> 2GB$
grover_s1	24	50	2301	94.2	—	$> 2GB$
grover_s_bf1	24	71	2208	94.3	—	$> 2GB$
grover_s_pf1	24	73	2258	94.2	—	$> 2GB$
bb84Eve	9	26	0.02	0.129	0.19	2.0
bb84NoEve	7	14	<0.01	0.0313	<0.01	0.152

Table 4.2: Performance results for QCSim and QuIDDPro on the benchmarks incorporating errors. $> 2GB$ indicates that a memory usage cutoff of 2GB was exceeded.

ranges from 5 to 20.

Performance results for these circuit benchmarks are shown in Table 4.3. Again, QuIDDPro has significantly better performance. These results highlight the fact that QCSim's explicit representation of the density matrix becomes an asymptotic bottleneck as n increases, while QuIDDPro's compression of the density matrix and operators scales extremely well.

4.5 Summary

We have described a new graph-based simulation technique that enables efficient density matrix simulation of quantum circuits. We implemented this technique in the QuIDDPro simulator. QuIDDPro uses the QuIDD data structure to compress redundancy in the gate operators and the density matrix. As a result, the time and memory complexity of QuIDDPro varies with the structure of the circuit. However, we demonstrated that QuIDDPro exhibits superior performance on a set of benchmarks which incorporate qubit

No. of Qubits	No. of Gates	QuIDDPro		QCSim	
		Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
5	32	0.05	0.0234	0.01	0.00781
6	50	0.07	0.0391	0.01	0.0352
7	84	0.11	0.043	0.08	0.152
8	126	0.16	0.0586	0.54	0.625
9	208	0.27	0.0742	3.64	2.50
10	324	0.42	0.0742	23.2	10.0
11	520	0.66	0.0898	151	40.0
12	792	1.03	0.105	933	160
13	1224	1.52	0.141	5900	640
14	1872	2.41	0.125	—	$> 2GB$
15	2828	3.62	0.129	—	$> 2GB$
16	4290	5.55	0.145	—	$> 2GB$
17	6464	8.29	0.152	—	$> 2GB$
18	9690	12.7	0.246	—	$> 2GB$
19	14508	18.8	0.199	—	$> 2GB$
20	21622	28.9	0.203	—	$> 2GB$

Table 4.3: Performance results for QCSim and QuIDDPro on the Grover's quantum search benchmark. $> 2GB$ indicates that a memory usage cutoff of 2GB was exceeded.

errors, mixed states, error correction, quantum communication, and quantum search. This result indicates that there is a great deal of structure in *practical* quantum circuits that graph-based algorithms like those implemented in QuIDDPro exploit.

CHAPTER V

Checking Equivalence of States, Operators and Circuits

A large body of work has developed around classical synthesis of quantum circuits [5, 61, 67]. Equivalence-checking of digital circuits is a key task in classical synthesis and verification, and is likely to be as important in quantum CAD, where equivalence checking of states and operators is more difficult. Unlike classical circuits, qubits and quantum gates may differ by global and relative phase yet be equivalent upon measurement [51]. Building upon the algorithmic blocks developed in Chapter III, we present a number of QuIDD algorithms which perform equivalence checking for states and operators which appear in [81]. Solutions to this problem have not been explored very much in the literature, and as we show, the variety of algorithms which do solve the problem is surprising.

The next section offers motivation for why equivalence checking is useful in quantum CAD. Section 5.2 describes both linear-algebraic and QuIDD algorithms for checking global-phase equivalence of states and operators. Section 5.3 covers relative-phase equivalence checking algorithms. Sections 5.2 and 5.3 also contain empirical studies comparing the algorithms' performance on various benchmarks. Lastly, a discussion of the results and a summary of computational complexities for all algorithms are provided in Section 5.4.

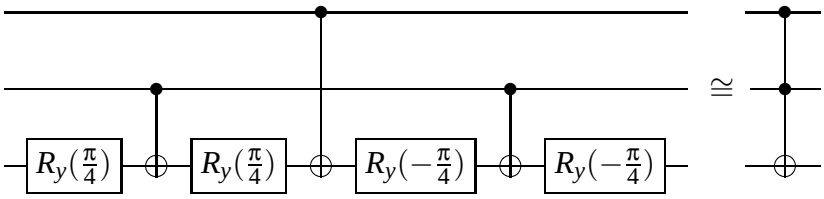


Figure 5.1: Margolus' circuit, which is equivalent up to relative phase to the Toffoli gate.

5.1 Motivation for Equivalence Checking

The extended notion of quantum equivalence creates several design opportunities. For example, the Toffoli gate can be implemented with fewer controlled-NOT (CNOT) and 1-qubit gates up to relative phase [5, 67] as shown in Figure 5.1. Normally the Toffoli gate requires six CNOT and eight 1-qubit gates to implement. The relative-phase differences induced can be canceled out so long as every pair of these gates in the circuit is strategically placed [67]. Since circuit minimization is being pursued for a number of key quantum arithmetic circuits with many Toffoli gates such as modular exponentiation [73, 26, 62, 61], this type of phase equivalence could reduce the number of gates even further.

Recall that two states $|\psi\rangle$ and $|\varphi\rangle$ are equal up to global phase if $|\varphi\rangle = e^{i\theta}|\psi\rangle$, where $\theta \in \mathbb{R}$. $e^{i\theta}$ will not be observed upon measurement of either state [51]. In contrast, two states are equal up to relative phase if

$$|\varphi\rangle = \begin{bmatrix} e^{i\theta_0} & & & \\ & e^{i\theta_1} & & \\ & & \ddots & \\ & & & e^{i\theta_{N-1}} \end{bmatrix} |\psi\rangle\,. \tag{5.1}$$

The probability amplitudes of the state $U|\psi\rangle$ will in general differ by more than relative

phase from those of $U|\varphi\rangle$, but the measurement outcomes may be the same. Global phase equivalence may be viewed as a special case of relative-phase equivalence in which all $e^{i\theta_j}$ are equal. Furthermore, identical states may be considered a special case of global-phase equivalence in which the phase factor is 1. Thus, the equivalence checking problem may be viewed as an equivalence hierarchy in which exact equivalence implies global-phase equivalence, which implies relative-phase equivalence, which in turn implies measurement outcome equivalence. The equivalence checking problem is also extensible to quantum operators with applications to quantum-circuit synthesis and verification, which involves computer-aided generation of minimal quantum circuits with correct functionality.

The inner product and matrix product may be used to determine such equivalences, but in this work, we present QuIDD algorithms to accomplish the task more efficiently. The algorithms solve the equivalence-checking problem asymptotically faster in some cases. Empirical results confirm the algorithms' effectiveness and show that the improvements are more significant for operators than for states. Interestingly, solving the equivalence problems for the benchmarks considered requires significantly less time than creating the QuIDD representations, which indicates that such problems can be reasonably solved in practice using quantum-circuit CAD tools.

5.2 Checking Equivalence up to Global Phase

This section describes algorithms that check global-phase equivalence of two quantum states or operators. The first two algorithms are well-known linear-algebraic operations, while the remaining algorithms exploit QuIDD properties explicitly. The section concludes with experiments comparing all algorithms.

5.2.1 Inner Product

Since the quantum-circuit formalism models an arbitrary quantum state $|\psi\rangle$ as a unit vector, the inner product $\langle\psi|\psi\rangle = 1$. In the case of a global-phase difference between two states $|\psi\rangle$ and $|\varphi\rangle$, the inner product is the global-phase factor, $\langle\varphi|\psi\rangle = e^{i\theta}\langle\psi|\psi\rangle = e^{i\theta}$. Since $|e^{i\theta}| = 1$ for any θ, checking if the complex modulus of the inner product is 1 suffices to check global-phase equivalence for states.

Although the inner product may be computed using explicit arrays, a straightforward QuIDD-based implementation is easily derived. The complex-conjugate transpose and matrix product with QuIDD operands have been previously defined in Chapter IV. Thus, the algorithm computes the complex-conjugate transpose of A and multiplies the result with B. The complexity of this algorithm is given by the following lemma.

Lemma 5.2 *Consider two state QuIDDs A and B with sizes $|A|$ and $|B|$, respectively, in number of nodes. The global-phase difference, if any, can be computed in $O(|A||B|)$ time and memory.*

Proof. Computing the complex-conjugate transpose of A requires $O(|A|)$ time and memory (Subsection 3.1.5). Matrix multiplication of two ADDs of sizes $|A|$ and $|B|$ requires $O((|A||B|)^2)$ time and memory (Subsection 3.1.4). However, this bound is loose for an inner product because only a single dot product must be performed. In this case, the ADD matrix multiplication algorithm reduces to a single call of $C = \mathbf{Apply}(A, B, *)$ followed by $D = \mathbf{Apply}(C, +)$ [4]. D is a single terminal node containing the global-phase factor if $|value(D)| = 1$. $\mathbf{Apply}(A, B, *)$ and $\mathbf{Apply}(C, +)$ are computed in $O(|A||B|)$ time and memory [17], while $|value(D)|$ is computed in $O(1)$ time and memory. □

5.2.2 Matrix Product

The matrix product of two operators can be used for global-phase equivalence checking. In particular, since all quantum operators are unitary, the adjoint of each operator is its inverse. Thus, if two operators U and V differ by a global phase, then $UV^\dagger = e^{i\theta}I$.

With QuIDD representations of U and V, computing $V^\dagger$ requires $O(|V|)$ time and memory (Subsection 3.1.5). The matrix product $W = UV^\dagger$ requires $O((|U||V|)^2)$ time and memory (Subsection 3.1.4). To check if $W = e^{i\theta}I$, any terminal value t is chosen from W, and scalar division is performed on W as $W' = \textbf{Apply}(W, t, /)$ which takes $O((|U||V|)^2)$ time and memory. Since QuIDDs are canonical, checking if $W' = I$ requires only $O(1)$ time and memory. If $W' = I$, then t is the global-phase factor.

5.2.3 Node-Count Check

The previous algorithms merely translate linear-algebraic operations to QuIDDs, but exploiting the following QuIDD property leads to faster checks.

Lemma 5.3 *The QuIDD $A' = \textbf{Apply}(A, c, *)$, where $c \in \mathbb{C}$ and $c \neq 0$, is isomorphic to A, hence $|A'| = |A|$.*

Proof. In creating A', **Apply** expands all of the internal nodes of A since c is a scalar, and the new terminals are the terminals of A multiplied by c. All terminal values t_i of A are unique by definition of a QuIDD (see Chapter III). Thus, $ct_i \neq ct_j$ for all i, j such that $i \neq j$. As a result, the number of terminals in A' is the same as in A. □

Lemma 5.3 states that two QuIDD states or operators that differ by a non-zero scalar, such as a global-phase factor, have the same number of nodes. Thus, equal node counts

```
GPRC(A,B,gp,have_gp) {
  if (Is_Constant(A) and Is_Constant(B)) {
    if (Value(B) == 0)  return (Value(A) == 0);
    ngp = Value(A)/Value(B);
    if (sqrt(real(ngp) * real(ngp)+
       imag(ngp) * imag(ngp)) != 1)
         return false;
    if (!have_gp) {
       gp = ngp;
       have_gp = true;
    }
    return (ngp == gp);
  }
  if ((Is_Constant(A) and !Is_Constant(B))
    or (!Is_Constant(A) and Is_Constant(B)))
       return false;
  if (Var(A)! = Var(B)) return false;
  return (GPRC(Then(A),Then(B),gp,have_gp)
   and GPRC(Else(A),Else(B),gp,have_gp));
}
```

Figure 5.2: Pseudo-code for the recursive global-phase equivalence check.

in QuIDDs are a necessary but not sufficient condition for global-phase equivalence. To see why it is not sufficient, consider two state vectors $|\psi\rangle$ and $|\varphi\rangle$ with elements w_j and v_k, respectively, where $j,k = 0,1,\ldots N-1$. If some $w_j = v_k = 0$ such that $j \neq k$, then $|\varphi\rangle \neq e^{i\theta}|\psi\rangle$. The QuIDD representations of these states can in general have the same node counts. Despite this drawback, the node-count check requires only $O(1)$ time since **Apply** is easily augmented to recursively sum the number of nodes as a QuIDD is created.

5.2.4 Recursive Check

Lemma 5.3 implies that a QuIDD-based algorithm which takes into account terminal value differences implements a sufficient condition for checking global-phase equivalence. The pseudo-code for such an algorithm called **GPRC** is presented in Figure 5.2.

GPRC returns **true** if two QuIDDs A and B differ by global phase and **false** otherwise.

gp and *have_gp* are global variables containing the global-phase factor and a flag signifying whether or not a terminal node has been reached, respectively. The value of *gp* is only valid if **true** is returned.

The first conditional block of **GPRC** deals with terminal values. The potential global-phase factor *ngp* is computed after handling division by 0. If $|ngp| \neq 1$ or if $ngp \neq gp$ when *gp* has been set,then the two QuIDDs do not differ by a global phase. Next, the condition specified by Lemma 5.3 is addressed. If the node of *A* depends on a different row or column variable than the node of *B*, then *A* and *B* are not isomorphic and thus cannot differ by global phase. Finally, **GPRC** is called recursively, and the results of these calls are combined via the logical *AND* operation.

Early termination occurs when isomorphism is violated or more than one phase difference is computed. In the worst case, both QuIDDs will be isomorphic, but the last terminal visited in each QuIDD will differ by more than a global-phase factor, causing full traversals of both QuIDDs. Thus, the overall runtime and memory complexity of **GPRC** for states or operators is $O(|A| + |B|)$. Also, the node-count check can be run before **GPRC** to quickly eliminate many nonequivalences.

5.2.5 Empirical Results for Global-Phase Equivalence

The first benchmark considered is a single iteration of Grover's quantum search algorithm [33], which is depicted in Figure 5.3. As in Chapter III, the oracle searches for the last item in the database. One iteration is sufficient to test the effectiveness of the algorithms since the state vector QuIDD remains isomorphic across all iterations, as proven in Subsection 3.2.2.

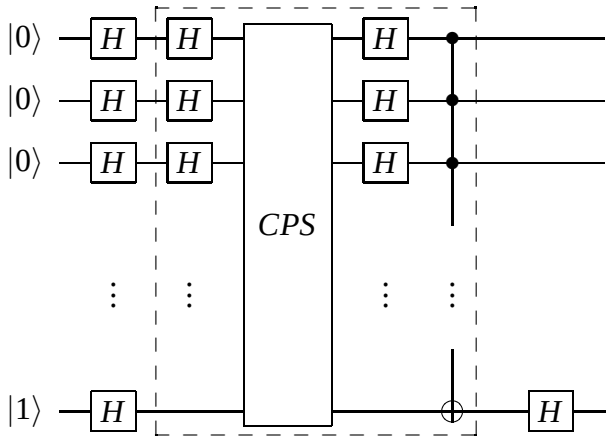
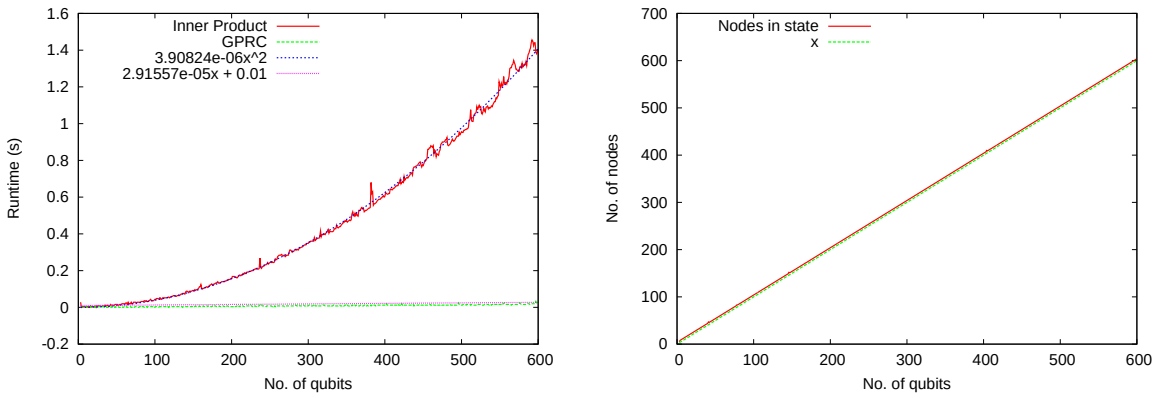


Figure 5.3: One iteration of Grover's search algorithm with an ancillary qubit used by the oracle. *CPS* is the conditional phase shift operator, while the boxed portion is the Grover iteration operator.

Figure 5.4a shows the runtime results for the inner product and **GPRC** algorithms (no results are given for the node-count check algorithm since it runs in $O(1)$ time). The results confirm the asymptotic complexity difference between the inner product and the **GPRC** algorithm. The number of nodes in the QuIDD state vector after a Grover iteration is $O(n)$ [80], which is confirmed in Figure 5.4b. As a result, the runtime complexity of the inner product should be $O(n^2)$, which is confirmed by a regression plot within 1% error. In contrast, the runtime complexity of the **GPRC** algorithm should be $O(n)$, which is also confirmed by another regression plot within the same error.

Figure 5.5a shows runtime results for the matrix product and **GPRC** algorithms checking the Grover operator. We showed in Chapter III that the QuIDD representation of this operator grows in size as $O(n)$, which is confirmed in Figure 5.5b. Therefore, the runtime of the matrix product should be quadratic in n but linear in n for **GPRC**. Regression plots verify these complexities within 0.3% error.



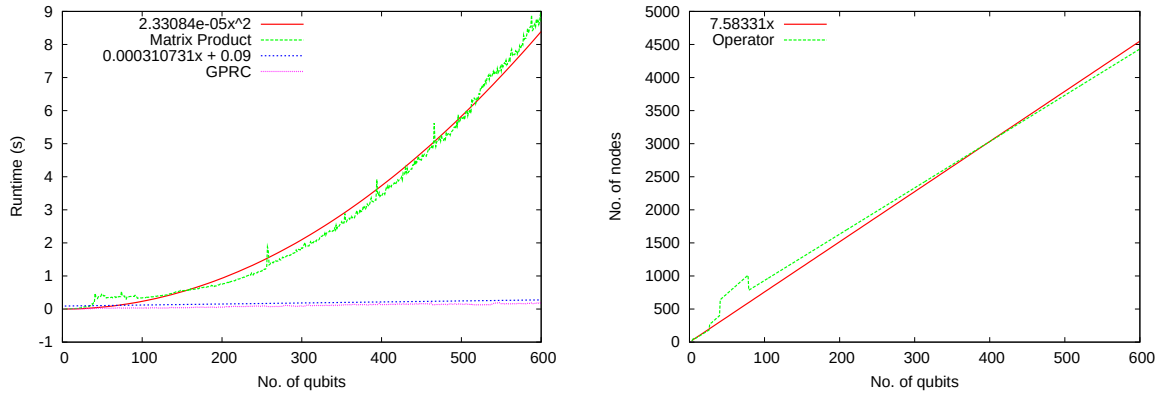
(a) (b)

Figure 5.4: (a) Runtime results and regressions for the inner product and **GPRC** on checking global-phase equivalence of states generated by a Grover iteration. (b) Size in node count and regression of the QuIDD representation of the state vector.

The next benchmark compares states that appear in Shor's integer factorization algorithm [65]. In particular, we consider states created by the modular exponentiation subcircuit that represent all possible combinations of x and $f(x,N) = a^x \ mod \ N$, where N is the integer to be factored (see Figure 5.6). Each of the $O(2^n)$ paths to a non-0 terminal encodes a binary value for x and $f(x,N)$. This benchmark demonstrates how the algorithms fare with exponentially-growing QuIDDs.

Tables 5.1a-d show the results of the inner product and **GPRC** for this benchmark. Each N is an integer whose two non-trivial factors are prime.[1] a is set to $N-2$ since it may be chosen randomly from the range $[2..N-2]$. In the case of Table 5.1a, states $|\psi\rangle$ and $|\varphi\rangle$ are equal up to global phase. The node counts for both states are equal as predicted by Lemma 5.3. Interestingly, both algorithms exhibit nearly the same performance. Tables 5.1b-d contain results for the cases in which Hadamard gates are applied to the first,

[1] Such integers are likely to be the ones input to Shor's algorithm since they are the foundation of modern public key cryptography [65].



(a) (b)

Figure 5.5: (a) Runtime results and regressions for the matrix product and **GPRC** on checking global-phase equivalence of the Grover iteration operator. (b) Size in node count and regression of the QuIDD representation of the operator.

middle, and last qubits, respectively, of $|\varphi\rangle$. These results show that early termination in **GPRC** can enhance performance by factors of roughly 1.5x to 10x.

In almost every case, both algorithms represent far less than 1% of the total runtime. Thus, checking for global-phase equivalence among QuIDD states appears to be an easily achievable task once the QuIDDs are created. An interesting side note is that in modular exponentiation, some QuIDD states with more qubits have more exploitable structure than those with fewer qubits. For instance, the $N = 387929$ (19 qubits) QuIDD has fewer than half the nodes of the $N = 163507$ (18 qubits) QuIDD.

Table 5.2 contains results for the matrix product and **GPRC** algorithm checking the inverse QFT operator. As noted in Chapter III, the inverse QFT is a key operator in Shor's algorithm [65], and its n-qubit QuIDD representation grows as $O(2^{2n})$. In this case, the asymptotic differences in the matrix product and **GPRC** are very noticeable. Also, the memory usage indicates that the matrix product may need asymptotically more intermedi-

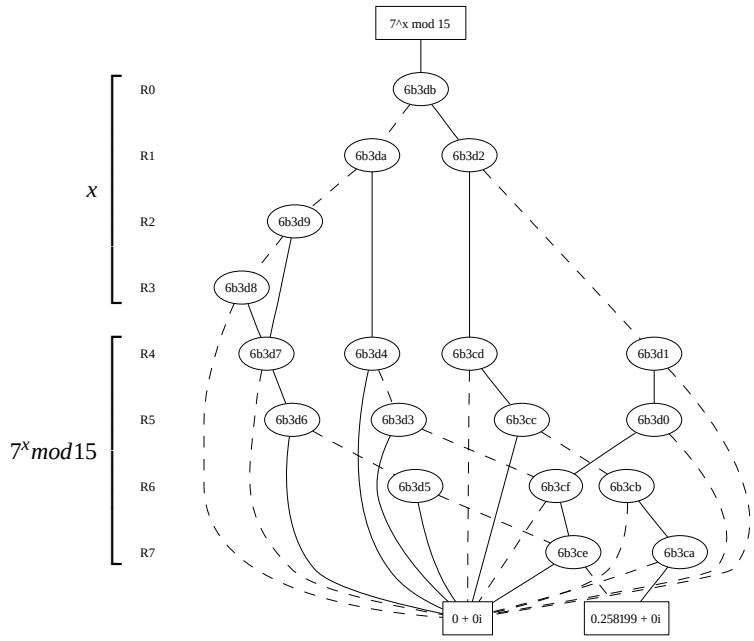


Figure 5.6: A QuIDD state combining x and $7^x \ mod\ 15$ in binary. The first qubit of each partition is least-significant. Internal node labels are unique hexadecimal identifiers based on each node's memory address with the variable depended upon listed to the left.

ate memory despite operating on QuIDDs with the same number of nodes as **GPRC**.

5.3 Checking Equivalence up to Relative Phase

Like global-phase equivalence, the relative-phase checking problem can be solved in several ways. The first three algorithms adapt standard linear algebra to QuIDDs, while the last two algorithms exploit QuIDD properties directly, offering asymptotic runtime and memory improvements.

5.3.1 Modulus and Inner Product

Consider two state vectors $|\psi\rangle$ and $|\varphi\rangle$ that are equal up to relative phase and have complex-valued elements w_j and v_k, respectively, where $j,k = 0,1,\ldots,N-1$. Comput-

No. of Qubits	N	Creation Time (s)	No. of Nodes $\|\psi\rangle$	No. of Nodes $\|\varphi\rangle$	IP Time (s)	GPRC Time (s)
10	993	2.37	273	273	0.012	0.008
11	1317	3.23	1710	1710	0.064	0.048
12	4031	11.9	9391	9391	0.30	0.26
13	6973	24.8	10680	10680	0.34	0.28
14	12127	55.1	18236	18236	0.54	0.46
15	19093	128.3	12766	12766	0.41	0.32
16	50501	934.1	51326	51326	1.7	1.6
17	69707	1969	26417	26417	0.87	0.78
18	163507	12788	458064	458064	19.6	19.6
19	387929	93547	182579	182579	6.62	6.02

(a)

No. of Nodes $\|\varphi\rangle$	IP Time (s)	GPRC Time (s)
508	0.012	$< 1e-10$
1812	0.052	0.004
10969	0.27	0.036
11649	0.31	0.036
19978	0.54	0.06
13446	0.41	0.036
55447	1.53	0.2
27797	0.78	0.084
521725	19.0	9.18
194964	6.44	4.40

(b)

No. of Qubits	N	Creation Time (s)	No. of Nodes $\|\psi\rangle$	No. of Nodes $\|\varphi\rangle$	IP Time (s)	GPRC Time (s)
10	993	2.37	273	508	0.016	$< 1e-10$
11	1317	3.23	1710	2768	0.068	0.024
12	4031	11.9	9391	11773	0.27	0.076
13	6973	24.8	10680	16431	0.43	0.14
14	12127	55.1	18236	29584	0.65	0.22
15	19093	128.3	12766	19207	0.56	0.20
16	50501	934.1	51326	71062	1.76	0.84
17	69707	1969	26417	46942	1.24	0.55
18	163507	12788	458064	653048	31.7	26.1
19	387929	93547	182579	312626	9.33	6.44

(c)

No. of Nodes $\|\varphi\rangle$	IP Time (s)	GPRC Time (s)
508	0.008	0.004
2768	0.056	0.008
14092	0.21	0.088
16431	0.27	0.084
29584	0.53	0.13
19207	0.50	0.084
74919	1.51	0.66
46942	1.13	0.25
629533	29.6	23.7
312626	13.0	8.62

(d)

Table 5.1: Performance results for the inner product and **GPRC** algorithms on checking global-phase equivalence of modular exponentiation states. In (a), $|\psi\rangle = |\varphi\rangle$ up to global phase. In (b), (c), and (d), Hadamard gates are applied to the first, middle, and last qubits of $|\varphi\rangle$ so that $|\psi\rangle \neq |\varphi\rangle$ up to global phase.

ing $|\varphi'\rangle = \Sigma_{i=0}^{N-1}|v_j|\,|j\rangle$ and $|\psi'\rangle = \Sigma_{k=0}^{N-1}|w_k|\,|k\rangle = \Sigma_{k=0}^{N-1}|e^{i\theta_k}v_k|\,|k\rangle$ sets each phase factor to 1, allowing the inner product to be applied as in Subsection 5.2.1. The complex modulus operations are computed as $C = \mathbf{Apply}(A, |\cdot|)$ and $D = \mathbf{Apply}(B, |\cdot|)$ with runtime and memory complexity $O(|A| + |B|)$, which is dominated by the $O(|A||B|)$ inner product complexity.

5.3.2 Modulus and Matrix Product

For operator equivalence up to relative phase, two cases are considered, namely the diagonal relative-phase matrix appearing on the left or right side of one of the operators. Consider two operators U and V with elements $u_{j,k}$ and $v_{j,k}$, respectively, where $j,k =$

No. of Qubits	Matrix Product		GPRC	
	Time (s)	Mem (MB)	Time (s)	Mem (MB)
3	0.036	0.13	0.004	0.13
4	0.30	0.39	0.016	0.13
5	2.53	1.41	0.064	0.25
6	22.55	6.90	0.24	0.66
7	271.62	46.14	0.98	2.03
8	3637.14	306.69	4.97	7.02
9	22717	1800.42	17.19	26.48
10	—	$> 2GB$	75.38	102.4
11	—	$> 2GB$	401.34	403.9

Table 5.2: Performance results for the matrix product and **GPRC** algorithms on checking global-phase equivalence of the QFT operator used in Shor's factoring algorithm. $> 2GB$ indicates that a memory usage cutoff of 2GB was exceeded.

$0, \ldots N-1$. The two cases in which the relative-phase factors appear on either side of V are described as $u_{j,k} = e^{i\theta_j} v_{j,k}$ (left side) and $u_{j,k} = e^{i\theta_k} v_{j,k}$ (right side). In either case the the matrix product check discussed in Subsection 5.2.2 may be extended by computing the complex modulus without increasing the overall complexity. Note that neither this algorithm nor the modulus and inner product algorithm calculate the relative-phase factors.

5.3.3 Element-wise Division

Given the states discussed in Subsection 5.3.1, $w_k = e^{i\theta_k} v_k$, the operation w_k / v_j for each $j = k$ is a relative-phase factor, $e^{i\theta_k}$. The condition $|w_k / v_j| = 1$ is used to check if each division yields a relative phase. If this condition is satisfied for all divisions, the states are equal up to relative phase.

The QuIDD implementation for states is simply $C = \mathbf{Apply}(A, B, /)$, where **Apply** is augmented to avoid division by 0 and instead return 1 when two terminal values being compared equal 0, and return 0 otherwise. **Apply** can be further augmented to terminate early when $|w_j / v_i| \neq 1$. C is a QuIDD vector containing the relative-phase factors. If C

contains a terminal value of 0, then A and B do not differ by relative phase. Since a call to **Apply** implements this algorithm, the runtime and memory complexity are $O(|A||B|)$.

Element-wise division for operators is more complicated. For QuIDD operators U and V, $W = \mathbf{Apply}(U,V,/)$ is a QuIDD matrix with the relative-phase factor $e^{i\theta_j}$ along row j in the case of phases appearing on the left side, and along column j in the case of phases appearing on the right side. In the first case, all rows of W are identical, meaning that the support of W does not contain any row variables. Similarly, in the second case the support of W does not contain any column variables. A complication arises when 0 values appear in either operator. In such cases, the support of W may contain both variable types, but the operators may in fact be equal up to relative phase. Figure 5.11 presents an algorithm based on **Apply** which accounts for these special cases by using a sentinel value of 2 to mark valid 0 entries that do not affect relative-phase equivalence.[2] These entries are recursively ignored by skipping either row or column variables with sentinel children (S specifies row or column variables), which effectively fills copies of neighboring row or column phase values in their place in W. The algorithm must be run twice, once for each variable type. The size of W is $O(|U||V|)$ since it is created with a variant of **Apply**.

5.3.4 Non-0 Terminal Merge

A necessary condition for relative-phase equivalence is that zero-valued elements of each state vector appear in the same locations, as expressed by the following lemma.

Lemma 5.4 *A necessary but not sufficient condition for two states* $|\varphi\rangle = \Sigma_{j=0}^{N-1} v_j |j\rangle$ *and* $|\psi\rangle = \Sigma_{k=0}^{N-1} w_k |k\rangle$ *to be equal up to relative phase is that* $\forall v_j = w_k = 0,\ j = k$.

[2] Any sentinel value larger than 1 may be used since such values do not appear in the context of quantum circuits.

Proof. If $|\psi\rangle = |\varphi\rangle$ up to relative phase, $|\psi\rangle = \Sigma_{k=0}^{N-1} e^{i\theta_k} v_k |k\rangle$. Since $e^{i\theta_k} \neq 0$ for any θ, if any $w_k = 0$, then $v_j = 0$ must also be true where $j = k$. A counter-example proving insufficiency is $|\psi\rangle = (0, 1/\sqrt{3}, 1/\sqrt{3}, 1/\sqrt{3})^T$ and $|\varphi\rangle = (0, 1/2, 1/\sqrt{2}, 1/2)^T$. □

QuIDD canonicity may be exploited with this condition. Let A and B be the QuIDD representations of the states $|\psi\rangle$ and $|\varphi\rangle$, respectively. First compute $C = \textbf{Apply}(A, \lceil|\cdot|\rceil)$ and $D = \textbf{Apply}(B, \lceil|\cdot|\rceil)$, which converts every non-zero terminal value of A and B into a 1. Since C and D have only two terminal values, 0 and 1, checking if C and D are equal satisfies Lemma 5.4. Canonicity ensures this check requires $O(1)$ time and memory. The overall runtime and memory complexity of this algorithm is $O(|A| + |B|)$ due to the unary **Apply** operations. This algorithm can also be applied to operators since Lemma 5.4 also applies to $u_{j,k} = e^{i\theta_j} v_{j,k}$ (phases on the left) and $u_{j,k} = e^{i\theta_k} v_{j,k}$ (phases on the right) for operators U and V.

5.3.5 Modulus and DD Compare

A variant of the algorithm presented in Subsection 5.3.1 which also exploits the canonicity of QuIDDs provides an asymptotic improvement for checking a necessary and sufficient condition of relative-phase equivalence of states and operators. As in Subsection 5.3.1, compute $C = \textbf{Apply}(A, |\cdot|)$ and $D = \textbf{Apply}(B, |\cdot|)$. If A and B are equal up to relative phase, then $C = D$ since each phase factor becomes a 1. Canonicity again ensures this check requires $O(1)$ time and memory. Thus, the runtime and memory complexity of this algorithm is dominated by the unary **Apply** operations, giving $O(|A| + |B|)$.

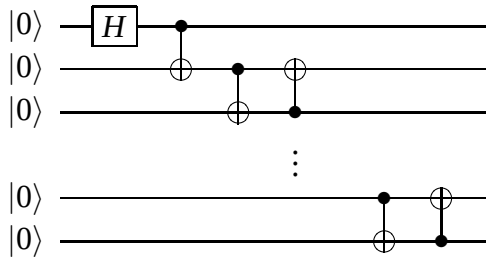


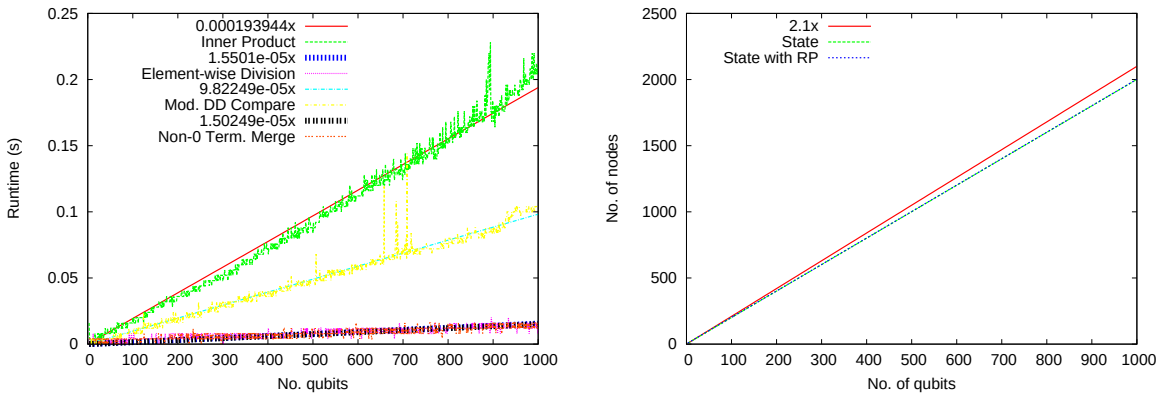
Figure 5.7: Remote EPR-pair creation between the first and last qubits via nearest-neighbor interactions.

5.3.6 Empirical Results for Relative-Phase Equivalence

We now present empirical results for the relative-phase equivalence checking algorithms. The first benchmark creates a remote EPR pair, which is an EPR pair between the first and last qubits, via nearest-neighbor interactions [11]. The circuit is shown in Figure 5.7 and is discussed in detail later in Chapter VI. Given an initial state $|00\ldots0\rangle$, it creates the remote EPR-pair state $(1/\sqrt{2})(|00\ldots0\rangle + |10\ldots1\rangle)$. The circuit size is varied, and the final state is compared to the state $(e^{0.345i}/\sqrt{2})\,|00\ldots0\rangle + (e^{0.457i}/\sqrt{2})\,|10\ldots1\rangle$.

Runtime results for all algorithms are provided in Figure 5.8a. The results show that all of the algorithms run quickly. For example, the inner product is the slowest algorithm, yet for a 1000-qubit instance it runs in approximately 0.2 seconds, a small fraction of the 7.6 seconds required to create the QuIDD state vectors.

Regressions of the runtime and memory data reveal linear complexity for all algorithms to within 1% error. This is not unexpected since the QuIDD representations of the states grow linearly with the number of qubits (see Figure 5.8b), and the complex modulus reduces the number of different terminals prior to computing the inner product. These



(a) (b)

Figure 5.8: (a) Runtime results and regressions for the inner product, element-wise division, modulus and DD compare, and non-0 terminal merge algorithms for checking relative-phase equivalence of the remote EPR pair circuit. (b) Size in node count and regressions of the QuIDD states compared.

results illustrate that in practice, the inner product and element-wise division algorithms can perform better than their worst-case complexity. Element-wise division should be preferred when QuIDD states are compact since unlike the other algorithms, it computes the relative-phase factors.

The Hamiltonian simulation circuit shown in Figure 5.9 is taken from [51, Figure 4.19, p. 210]. When its one-qubit gate (boxed) varies with Δt, it produces a variety of diagonal operators, all of which are equivalent up to relative phase. Empirical results for such equivalence checking are shown in Figure 5.10. As in the case of the teleportation circuit benchmark, the matrix product and element-wise division algorithms perform better than their worst-case asymptotic upper-bounds, indicating that element-wise division is the best choice for compact QuIDD operators.

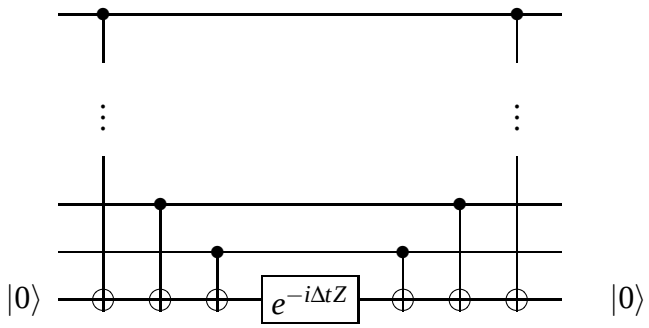
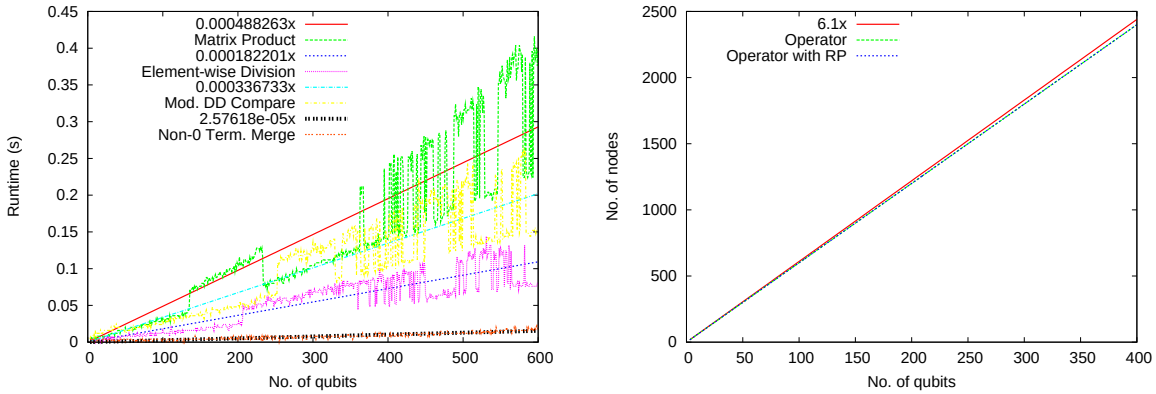


Figure 5.9: Quantum-circuit realization of a Hamiltonian consisting of Pauli operators. Extra Pauli gates may be needed depending on the Hamiltonian.



(a) (b)

Figure 5.10: (a) Runtime results and regressions for the matrix product, element-wise division, modulus and DD compare, and non-0 terminal merge algorithms for checking relative-phase equivalence of the Hamiltonian Δt circuit. (b) Size in node count and regressions of the QuIDD operators compared.

5.4 Summary

Although QuIDD properties like canonicity enable exact equivalence checking in $O(1)$ time, we have shown that such properties may be further exploited to develop efficient algorithms for the difficult problem of equivalence checking up to global and relative phase. In particular, the global-phase recursive check and element-wise division algorithms ef-

ficiently determine equivalence of states and operators up to global and relative phase, while computing the phases. In practice, they outperform QuIDD implementations of the inner and matrix product, which do not compute relative-phase factors. Other QuIDD algorithms presented in this work, such as the node-count check, non-0 terminal merge, and modulus and DD compare, further exploit DD properties to provide even faster checks but only satisfy necessary conditions for equivalence. Thus, they should be used to aid the more robust algorithms. A summary of the theoretical results presented in this paper is provided in Table 5.3.

The algorithms presented here enable QuIDDs to be used in synthesis and verification of quantum circuits, which was identified as the third goal of quantum circuit simulation. A fair amount of work has been done on optimal synthesis of small quantum circuits as well as heuristics for synthesis of larger circuits via circuit transformations [56, 61]. Equivalence checking in particular plays a key role in some of these techniques since it is often necessary to verify the correctness of the transformations. Future work will determine how these equivalence checking algorithms may be used as primitives to enhance such heuristics. Another interesting direction to explore is the use of density matrices for synthesis guided by error-based requirements and perhaps even mixed states. Such work would build upon the developments of Chapter IV in addition to the operator equivalence-checking algorithms described in this chapter.

Algorithm	Phase type	Finds phases?	Necessary & sufficient?	$O(\cdot)$ time complexity: best-case	$O(\cdot)$ time complexity: worst-case
Inner Product	Global	Yes	N. & S.	$\|A\|\|B\|$	$\|A\|\|B\|$
Matrix Product	Global	Yes	N. & S.	$(\|A\|\|B\|)^2$	$(\|A\|\|B\|)^2$
Node-Count	Global	No	N. only	1	1
Recursive Check	**Global**	**Yes**	**N. & S.**	**1**	$\|\mathbf{A}\|+\|\mathbf{B}\|$
Modulus and Inner Product	Relative	No	N. & S.	$\|A\|\|B\|$	$\|A\|\|B\|$
Element-wise Division	**Relative**	**Yes**	**N. & S.**	$\|\mathbf{A}\|\|\mathbf{B}\|$	$\|\mathbf{A}\|\|\mathbf{B}\|$
Non-0 Terminal Merge	Relative	No	N. only	$\|A\|+\|B\|$	$\|A\|+\|B\|$
Modulus and DD Compare	Relative	No	N. & S.	$\|A\|+\|B\|$	$\|A\|+\|B\|$

Table 5.3: Key properties of the QuIDD-based phase-equivalence checking algorithms.

```
RP_DIV(A,B,S) {
  if (A == New_Terminal(0)) {
    if (B != New_Terminal(0))
      return New_Terminal(0);
    return New_Terminal(2);
  }
  if (Is_Constant(A) and Is_Constant(B)) {
    nrp = Value(A)/Value(B);
    if (sqrt(real(nrp)*real(nrp)+
      imag(nrp)*imag(nrp)) != 1)
        return New_Terminal(0);
    return New_Terminal(nrp);
  }
  if (Table_Lookup(R,RP_DIV,A,B,S)) return R;
  v = Top_Var(A,B);
  T = RP_DIV(A_v,B_v,S);
  E = RP_DIV(A_v',B_v',S);
  if ((T == New_Terminal(0)) or
    (E == New_Terminal(0)))
      return New_Terminal(0);
  if ((T != E) and (Type(v) == S)) {
    if (Is_Constant(T) and Value(T) == 2)
      return E;
    if (Is_Constant(E) and Value(E) == 2)
      return T;
    return New_Terminal(0);
  }
  if (Is_Constant(T) and Value(T) == 2)
    T = New_Terminal(1);
  if (Is_Constant(E) and Value(E) == 2)
    E = New_Terminal(1);
  R = ITE(v,T,E);
  Table_Insert(R,RP_DIV,A,B,S);
  return R;
}
```

Figure 5.11: Pseudo-code for element-wise division algorithm.

CHAPTER VI

Further Speed-Up Techniques

This chapter describes a few ways to speed up QuIDD-based simulation that are captured by the QuIDDPro language. These techniques apply to QuIDD matrix multiplication, the tensor product, and the partial trace, which are key operations for simulation with and without errors. Section 6.1 describes algorithms for applying controlled- and 1-qubit gates to QuIDD state vectors. The simulator uses these algorithms when processing particular source-code expressions at the input. Section 6.2 demonstrates how the language may be used to selectively tensor and remove, via the partial trace over density matrices, qubits that do not affect the final outcome of a computation. Although this technique cannot be performed in every case, we focus on a circuit of interest in the literature for which this optimization exponentially reduces the asymptotic complexity of QuIDD-based simulation. This circuit is simulated with several types of random, continuous error effects. The effectiveness of "bang-bang" error correction is also studied.

6.1 Gate Algorithms

Matrix multiplication is the main operation for simulating quantum circuits as it provides the mathematical machinery for applying gates to qubits. QuIDDs utilize a variant

of the ADD-based matrix multiplication algorithm described in Subsection 3.1.4. This algorithm is a fairly straightforward translation of dot-products to the graph domain and makes use of the **Apply** algorithm [4]. It assumes that the two QuIDD arguments have the same dimension. A consequence of this assumption is that for small gates, say 1- and 2-qubit gates, a larger operator must be constructed by computing tensor products with identity matrices. For example, to apply a 1-qubit gate to some qubit of a 5-qubit state vector or density matrix, the matrix representing the gate must be first tensored with four 1-qubit identity matrices and then multiplied with the QuIDD representing the entire state vector or density matrix. It is natural to ask, however, if a more clever approach can be used to apply gates by leveraging the peculiar properties of QuIDDs, at least for certain types of gates. Indeed, such an improvement is possible as we now explain. Importantly, the QuIDDPro simulator automatically detects when this optimization may be performed through special expressions. The specialized algorithms are described first (Subsections 6.1.1 and 6.1.2), followed by a brief discussion of the relevant QuIDDPro language features (Subsection 6.1.3), and concluding with empirical results which demonstrate that the optimizations enable QuIDDs to be competitive with the stabilizer formalism (Subsection 6.1.4).

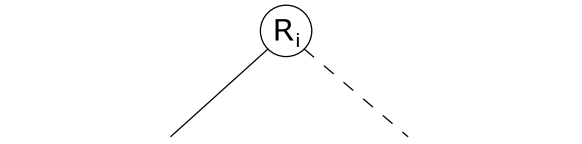
6.1.1 Simulating 1-qubit Gates

Special processing for small gates can be of great practical value considering that CNOT and all 1-qubit gates form a universal gate set [5]. The benefit of special processing for 1- and 2-qubit gates has been recognized previously, and is, in fact, the key notion underlying qubit-wise multiplication (Section 2.1). Unfortunately, qubit-wise multiplica-

$$UV = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}$$

$$= \begin{bmatrix} u_{00}v_0 + u_{01}v_1 \\ u_{10}v_0 + u_{11}v_1 \end{bmatrix}$$

(a)



(b)

Figure 6.1: (a) A 1-qubit gate applied to a single qubit, and (b) the QuIDD state vector transformation induced by this operation on qubit *i*.

tion only reduces the computational complexity of representing the operator and leaves the state vector or density matrix in an explicit, exponentially-sized form. A straightforward translation of the qubit-wise multiplication algorithm to QuIDDs would still result in exponential runtime since the algorithm iterates over all the indices of an array containing the state information.

Instead, an algorithm is needed which can both represent small operators concisely *and* update the state efficiently. An important property of QuIDDs is that each internal node R_i of a QuIDD state vector maps directly to qubit i since R_i represents the ith binary index of a state vector (Section 3.1). This means that applying a 1-qubit gate to qubit i can be accomplished simply by manipulating any instances of R_i nodes in a QuIDD state vector.

Given a QuIDD state vector, a top-down traversal is performed which transforms any R_i visited as shown in Figure 6.1. The transformation on R_i comes from the linear-algebraic description of a 1-qubit gate U acting on a 1-qubit state vector V to produce a new state vector V'. The probability amplitude for the $|0\rangle$ component of V' is $u_{00}v_0 + u_{01}v_1$. As a result, the subgraph pointed to by the 0 edge of the R_i node, or $E(R_i)$, is transformed into $u_{00}E(R_i) + u_{01}T(R_i)$. This operation is easily accomplished by two scalar multiplications via $E' = \mathbf{Apply}(E(R_i), u_{00}, *)$ and $T' = \mathbf{Apply}(T(R_i), u_{01}, *)$ followed by a single call to

Apply$(E', T', +)$ to add the two results. The same transformation is also performed on the subgraph pointed to by the 1 edge, except that u_{10} and u_{11} are used. If an R_i variable is missing in any particular path, which can be detected by encountering an R_j node such that $i < j$, then a new R_i node is created with children $u_{00}R_j + u_{01}R_j$ and $u_{10}R_j + u_{11}R_j$. Special checks on the node cache are performed to detect if the new children are equal, which results in the elimination of the R_i node as per the standard BDD rules. By performing this specialized 1-qubit gate operation on all R_i nodes in the QuIDD, the 1-qubit gate acting on qubit i need never be expanded into a larger n-qubit gate. All the extra overhead of the general ADD matrix multiplication algorithm is also avoided. Pseudo-code for this algorithm is provided in Figure 6.2.

6.1.2 Simulating Controlled Gates

A controlled-U gate can also be implemented more efficiently using the 1-qubit gate QuIDD algorithm. Suppose a controlled-U gate is applied with qubit i as the control and qubit j as the target, such that $i < j$. As before, a top-down traversal is performed, but when any R_i node is encountered, the traversal only continues down the 1 edge of the R_i node, or $T(R_i)$, since a standard controlled-U gate performs no action when the control qubit is a $|0\rangle$. After proceeding down the 1 edge of any R_i node, U is applied upon encountering any R_j node using the 1-qubit gate QuIDD algorithm. This operation is analogous to classical digital circuit simulation where the "controlling" values of logic gates are checked first before any other inputs [34]. For example, if an input wire of a k-input *OR* gate carries a 1 signal, then there is no need to check the other inputs since the output must be 1.

Interestingly, the controlled-U QuIDD algorithm is computationally more efficient

```
Q1_ALG(A, Op, qubit_index) {
  if (Table_Lookup(R, Q1_ALG, A, Op, qubit_index))
    return R;
  v = Var(A);
  if (Is_Constant(A) or Index(v) >= qubit_index) {
    if (Index(v) == qubit_index)) {
      T = A_v;
      E = A_v';
    }
    else T = E = A;
    E00 = Apply(E, New_Terminal(Op_0,0), *);
    T01 = Apply(T, New_Terminal(Op_0,1), *);
    E10 = Apply(E, New_Terminal(Op_1,0), *);
    T11 = Apply(T, New_Terminal(Op_1,1), *);
    E = Apply(E00, T01, +);
    T = Apply(E10, T11, +);
    R = ITE(v, T, E);
    Table_Insert(R, Q1_ALG, A, Op, qubit_index);
    return R;
  }
  T = Q1_ALG(A_v, Op, qubit_index);
  E = Q1_ALG(A_v', Op, qubit_index);
  R = ITE(v, T, E);
  Table_Insert(R, Q1_ALG, A, Op, qubit_index);
  return R;
}
```

Figure 6.2: Pseudo-code for the 1-qubit gate algorithm. $Op_{i,j}$ denotes accessing the complex value at row i and column j of the 1-qubit matrix Op.

than the 1-qubit gate QuIDD algorithm. The reason for this is simply that the controlled-U algorithm reduces the number of nodes in the QuIDD state vector that must be traversed by only traversing the 1 edges of controlling R_i nodes corresponding to control qubit i. Generalizing this result to controlled-U gates with multiple controls $i, i+1, \ldots, i+k$ such that $i+k < j$ shows that increasing the number of controls also increases the computational efficiency as each control further reduces the number of nodes to be traversed in the QuIDD state vector.

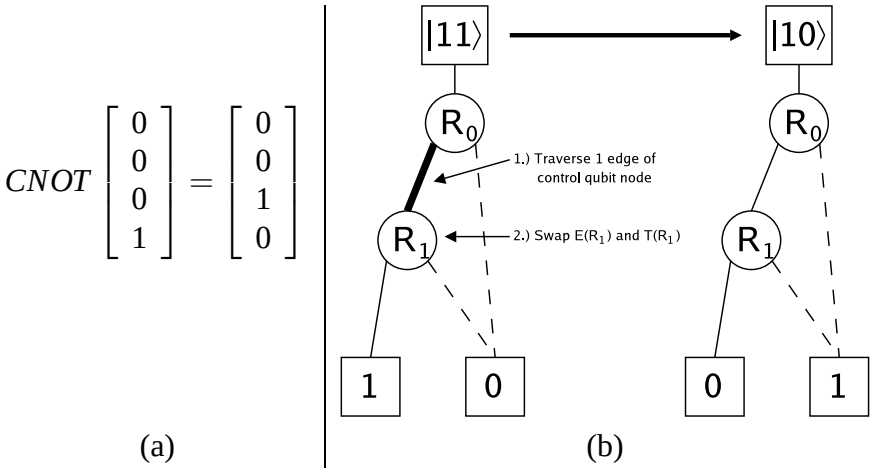


Figure 6.3: (a) A CNOT gate applied to the $|11\rangle$ state vector, and (b) the same operation applied using the specialized QuIDD algorithm.

A further improvement can be made in the specific case of the CNOT gate. The top-down traversal proceeds as before, but only down the 1 edges of R_i nodes. However, when an R_j node is reached, the $E(R_j)$ and $T(R_j)$ subgraphs are simply swapped instead of applying the *NOT* gate. The reason this can be done is because the action of a *NOT* gate is precisely to switch the amplitudes of the $|0\rangle$ and $|1\rangle$ components of a qubit. A simple example of this algorithm operating on the QuIDD state vector $|11\rangle$ is depicted in Figure 6.3.

An important point to note is that the specialized controlled-*U* QuIDD algorithm only considers the case in which the control qubit precedes the target qubit. The reason that a bottom-up traversal cannot be used to implement a controlled-*U* gate whose target may precede one or more controls is due to the sharing of nodes across QuIDDs. For any DD, nodes are shared within the DD and *across* multiple instances of such data structures. This sharing across DDs not only increases efficiency, but it's a requirement for proper functioning since efficient construction of any new DD through the **Apply** function requires

accessing the same node cache used by the DD arguments to **Apply** [17, 66]. As a result, there is no way for a bottom-up traversal to determine which DD it is in, since the terminal it starts at and any subsequent node it visits can be shared by multiple DDs. In contrast, a top-down traversal starts at the head of a specific DD.

If this is the case, then what can be done when a CNOT gate is applied to qubits i and j such that $i > j$? In this situation, the circuit equivalence for the "upside-down" CNOT is employed as shown in Figure 6.4a [51]. Using this equivalence, 1-qubit Hadamards can be applied using the specialized 1-qubit algorithm in conjunction with the specialized CNOT algorithm. This means that applying specialized controlled-U algorithms to QuIDDs is computationally more efficient when the control qubit precedes the target qubit.

For the general case in which multiple controls are preceded by the target, swap gates [51] can be employed to swap the target qubit with the last control qubit. As shown in Figures 6.4a and 6.4b, the swap gate can be implemented with CNOT and 1-qubit Hadamard gates. In QuIDDPro, however, a special DD function is used to swap nodes. This function has better performance by a large constant factor as compared to applying an actual swap gate since only one traversal of the QuIDD must be performed.

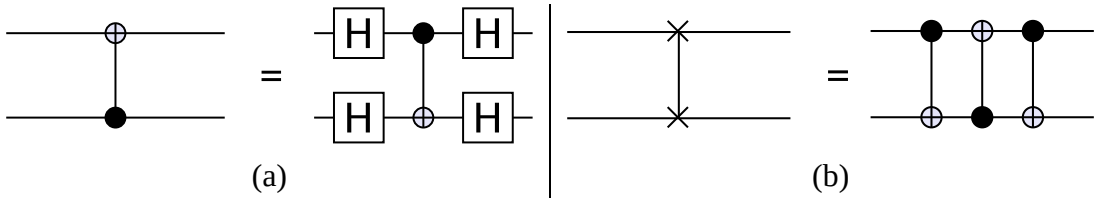


Figure 6.4: (a) A CNOT whose target precedes its control is shown next to an equivalent circuit composed of 1-qubit Hadamard gates and a CNOT with the control and target qubits reversed. (b) A swap gate, which exchanges the values of two qubits, shown next to an equivalent circuit composed of CNOT gates. The CNOT gate in the center can be converted as shown in (a).

6.1.3 Automatic Usage of Algorithms

In order for these algorithms to be used without putting the burden on the user of detecting the special cases, the simulator must know when to apply them. To this end, we introduced a new function to the QuIDDPro input language (see Appendix B) called *cu_gate*. This function uses a string to specify which qubits are controls (or negated controls) and targets. To demonstrate, suppose the user wants to apply a 1-qubit Pauli gate Y to qubits 4 and 7 conditional on control qubits 2 and 3, and negated control qubit 5. Further suppose that the total size of the circuit is 8 qubits. The *cu_gate* expression for this case is $cu\_gate(sigma\_y(1), \text{“}c2c3x4n5x7\text{”}, 8)$, where *c*, *n*, and *x* flag the subsequent qubit number as a control, negated control, or target, respectively. All unspecified qubits are assumed to be unaffected by the gate. Ordering within the specification string is irrelevant and handled internally by the simulator.

In the absence of the specialized gate-specific functions, QuIDDPro will create a QuIDD matrix according to the specifications given to *cu_gate*. This is accomplished efficiently with a series of tensor products and projections, all implemented with QuIDD algorithms. However, in the case of $cu\_gate(\cdot) * |state :>$, where $|state :>$ is a state vector QuIDD, the simulator does not create the QuIDD matrix. Instead, the specialized controlled-gate algorithm is performed on the state vector QuIDD directly. The specialized 1-qubit algorithm is applied when only “x’s” exist in the string specification (1-qubit gates are viewed as a special case of a controlled-gate with no controls).

Unfortunately, since QuIDDPro has a very expressive language[1], the situation becomes

[1] QuIDDPro has approximately 100 built-in functions and other language features, which are detailed in Appendix B.

more complicated when the result of a call to *cu_gate* is stored in a variable and applied at a later time to a state vector QuIDD. For example, consider $U = cu\_gate(\cdot)$ followed arbitrarily later in the QuIDDPro script by $U * |state :>$. To handle such cases, operators created with *cu_gate* are *lazily evaluated*. In other words, QuIDDPro associates matrix variables with control/target information and no QuIDD matrix is created for as long as possible. When the gate is multiplied with another gate or printed to standard output, the QuIDD matrix is created only at that point in the simulation. As a result, if gates are always applied to state vectors, any gates created with $cu\_gate(\cdot)$ are never actually created, and the faster algorithms described in the previous subsections are used instead. As we demonstrate next, this feature greatly enhances QuIDDPro's performance.

6.1.4 Empirical Results

We tested these specialized algorithms in QuIDDPro against the explicit ADD-based matrix multiplication algorithm. As evidenced by the results shown in Table 6.1, the specialized algorithms far outperform the matrix multiplication algorithm. "chp100" is a randomly generated 100-qubit circuit consisting of CNOT, Hadamard, and phase gates, which are the Clifford group generators (Section 2.5). "tchp100" is also a randomly generated 100-qubit circuit consisting of the Clifford group generators, but it also includes Toffoli gates. The addition of Toffoli gates is interesting since it forms a universal gate set for classical logic circuits [51]. "cnot200" stress tests the specialized controlled gate algorithm since it is a randomly generated 200-qubit circuit consisting only of CNOT gates. Similarly, "toff200" is a circuit of the same size but with Toffoli gates only. As evidenced by the results, the performance improvements are as large as $60\times$. This indicates that

the overhead avoided by specialization is significant. The results also demonstrate that the specialized algorithms allow QuIDDPro to simulate stabilizer circuits and stabilizer circuits with non-stabilizer Toffoli gates very efficiently for large circuit sizes, making QuIDDPro competitive in practice with the stabilizer formalism.

Benchmark	No. of Qubits	No. of Gates	Specialized Algorithms			ADD-based Multiplication		
			Runtime (s)	Avg. Time per Gate (s)	Memory (MB)	Runtime (s)	Avg. Time per Gate (s)	Memory (MB)
chp100	100	300	4.57	0.0152	9.85	48.9	0.163	5.18
tchp100	100	300	0.870	0.00290	4.51	10.8	0.0361	1.61
cnot200	200	1000	2.54	0.00254	7.14	125	0.125	6.93
toff200	200	1000	4.61	0.00461	7.20	154	0.154	9.30

Table 6.1: Performance results comparing QuIDDPro using the specialized algorithms to QuIDDPro using ADD-based matrix multiplication.

6.2 Dynamic Tensor Products and Partial Tracing

This section discusses other language features related to the density matrix model which enable QuIDDPro to efficiently simulate a particular circuit of interest in the presence of continuous, random errors. Normally the size of a QuIDD is sensitive to the number of different matrix elements (see Chapter III), but clever use of the QuIDDPro input language can reduce the negative effects of this sensitivity in certain cases. In particular, we dynamically add qubits to a density matrix state via the tensor product and removing them when they no longer affect the simulation results by tracing over them (Subsection 6.2.1). The benchmark circuit, the error model used, and empirical results are also discussed in the following subsections. The results include characterizations of imperfect gate errors, systematic errors, decoherence, and “bang-bang” error correction.

6.2.1 Language Support

In general, when some density state ρ_1 is not entangled with another state ρ_2, then $\rho_1 = tr_{\rho_2}(\rho_1 \otimes \rho_2)$ and similarly $\rho_2 = tr_{\rho_1}(\rho_1 \otimes \rho_2)$. In the course of simulation, if the qubits described by ρ_2 no longer affect the final states of interest, then they may be traced out to reduce simulation complexity. Rather than hold on to the separated state ρ_2 as in *p*-blocked simulation, it may be discarded entirely.

As will be shown in the next subsection, circuits with nearest-neighbor interactions tend to contain qubits that play only fleeting roles in affecting the qubits of interest. The purpose behind this technique is to introduce such qubits to the state only at the moment they are needed and to eliminate them from the state (and from memory) the moment they are no longer needed.

Since the QuIDDPro language provides a linear-algebraic interface, QuIDD matrices representing qubit states my be tensored at any time with density states of new qubits (see Appendix B). Furthermore, the partial trace may be efficiently performed at any time on any desired qubit (see Chapter IV). Although this technique is not automated for the user, these language features allow the user to very easily implement the optimization at various points in circuits whose functionality is well-understood. We now describe one such circuit which is used as a case study to demonstrate the effectiveness of this technique.

6.2.2 Motivation for Error Characterization

Some error-correcting code techniques have been developed to cope with errors in quantum hardware [69, 20, 29], but they require extra qubits and are most effective in the presence of single qubit errors only. Since the addition of extra qubits can be a daunting

technological task, it can be very helpful to know a priori if error effects will be significant enough to require such correction. A different error correction approach has been proposed which involves applying corrective "$2\pi k$" pulses without the need for additional qubits or the single qubit error constraint [9]. The effectiveness of this technique has been demonstrated in the context of teleporting qubits in nuclear spin chain quantum computers via remote entanglement [2] achieved by nearest-neighbor interactions [11].

Although the aforementioned work specifically considers nuclear spin quantum computing, remote entanglement through nearest-neighbor interaction is a common phenomenon in a number of other potential quantum computing technologies [89, 44, 55, 71, 15]. In the case of ion traps, even though qubits can be physically moved around, once in place, qubit interactions are performed between neighboring ions [41]. Equally important is the development of bang-bang error correction techniques which are a generalization of corrective pulses that decouple qubits from the environment, delaying the negative effects of decoherence error for any technology [87, 46, 86, 39, 57]. As a result, simulating the effect of error in remote entanglement achieved by nearest-neighbor interactions using the technology-independent quantum circuit model is an appealing case study.

6.2.3 Remote Entanglement Circuits

Remote entanglement enables teleportation of an arbitrary quantum state from one party to another. The key ingredient in this scheme is the creation of an EPR pair between two communicating parties, Alice and Bob, as described in Equation 1.17 of Section 1.2.1. Recall that the utility of this state lies in the fact that if Alice measures her particle and

[2] Remote entanglement refers to any entanglement between qubits that are not nearest neighbors.

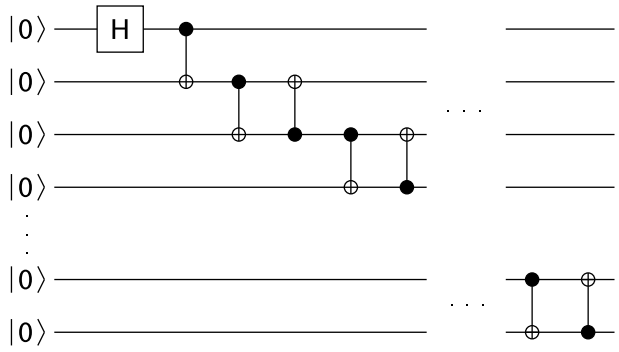


Figure 6.5: The remote EPR pair generation circuit which creates an EPR pair between qubits 0 (the top qubit) and $n-1$ (the bottom qubit) via nearest-neighbor interactions. The gate notation used comes from [51]. There are $2n-2$ gates in the circuit.

obtains a $|0\rangle$, then Bob will subsequently also obtain a $|0\rangle$ upon measurement of his particle. With only two qubits in the ground state, an EPR pair can be created by applying a Hadamard gate followed by a CNOT gate [11, 51, 8],

$$\Psi = (CNOT)(H \otimes I)\,|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{6.1}$$

In a circuit with more than two qubits, the above procedure can be generalized using nearest-neighbor interactions to create an EPR pair between qubits 0 and $n-1$ only. One straightforward generalization is to use a known nearest-neighbor decomposition of a CNOT gate with qubit 0 as the control and qubit $n-1$ as the target. Such a CNOT gate can be decomposed into $4(n-1)$ nearest-neighbor CNOT gates [60, Figure 3]. However, by making use of the fact that all qubits are initialized to the ground state, a smaller decomposition can be achieved with only $2n-3$ CNOT gates [11]. This circuit is shown in Figure 6.5 and generates the state

$$\Psi_R = \frac{1}{\sqrt{2}}(|00\ldots0\rangle + |10\ldots1\rangle). \tag{6.2}$$

In this state, qubits 0 and $n-1$ are remotely entangled since the measurement outcome of one qubit affects the measurement outcome of the other, yet the qubits are not neighbors. The circuit creates a remotely entangled EPR pair in the following way. The Hadamard gate and first CNOT gate create an EPR pair between qubits 0 and 1, just as in the 2-qubit case (Equation 6.1). The second CNOT gate creates an EPR "triple" $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ on the first three qubits. The third CNOT gate eliminates qubit 1 from the triple, leaving qubits 0 and 2 in an EPR pair, $\frac{1}{\sqrt{2}}(|000\rangle + |101\rangle)$. By induction, each subsequent pair of CNOT gates first creates an EPR triple among qubits 0, i, and $i+1$, and then removes qubit i from the triple, leaving qubits 0 and $i+1$ in an EPR pair. In this fashion, a remotely entangled EPR pair is eventually created among qubits 0 and $n-1$ via nearest-neighbor interactions.

In the absence of errors, the two computational basis states $|00\ldots0\rangle$ and $|10\ldots1\rangle$ occur upon measurement with a probability of $\frac{1}{2}$. All other states occur with probability 0. In the presence of errors described in Subsection 6.2.4, the probabilities of the two desired states will become less than $\frac{1}{2}$, and the probabilities of the other undesired states will become greater than 0.

6.2.4 Error Model

We first model random continuous gate error. The physical basis for this error is imprecision in the method used to apply gates to qubits. The implementation of gates for most known quantum computing technologies involves manipulation of electro-magnetic (EM) radiation pulses, and the quantum control imprecision for these pulses can manifest in under- or over-rotation of qubits [50, 41, 20, 9, 11, 89, 44, 55, 71]. As a result, our error

model has the general form of a 1-qubit unitary matrix,

$$U(\theta) = \begin{bmatrix} cos(\theta/2) & -sin(\theta/2) \\ sin(\theta/2) & cos(\theta/2) \end{bmatrix}, \tag{6.3}$$

where θ is a rotation parameter which depends directly on the desired EM frequency [51]. Not shown are different phase factors in front of each of the four elements. These factors are easily set appropriately depending on the type of gate that must be applied [51]. Modeling a faulty 1-qubit gate with continuous error only requires the addition of a small random ε error parameter to θ,

$$U_f(\theta,\varepsilon) = \begin{bmatrix} cos(\theta/2+\varepsilon) & -sin(\theta/2+\varepsilon) \\ sin(\theta/2+\varepsilon) & cos(\theta/2+\varepsilon) \end{bmatrix}, \tag{6.4}$$

where ε is normally distributed about 0 with standard deviation σ [29]. This model for continuous gate error was used to study the effectiveness of error correction codes in nearest-neighbor qubit arrays. It was shown that for such error correcting codes to be most effective, the ε error must range between 10^{-5} and 10^{-7} [29]. Also, in the nearest-neighbor nuclear-spin chain setting, an ε of around 10^{-6} is considered reasonable [9, 11]. The 1-qubit continuous gate error model can be extended to 2-qubit controlled gates as follows,

$$V_f(\theta_0,\theta_1,\varepsilon_0,\varepsilon_1) = |0\rangle\langle 0| \otimes U_0(\theta_0,\varepsilon_0) + |1\rangle\langle 1| \otimes U_1(\theta_1,\varepsilon_1), \tag{6.5}$$

where U_0 is a faulty gate describing the action on the control qubit, and U_1 is a faulty gate describing the action on the target qubit. In the case of a faulty CNOT, U_0 is a faulty identity gate, and U_1 is a faulty X gate [51]. To reverse the order of the control and target qubits, the operands of the tensor products in Equation 6.5 are simply reversed.

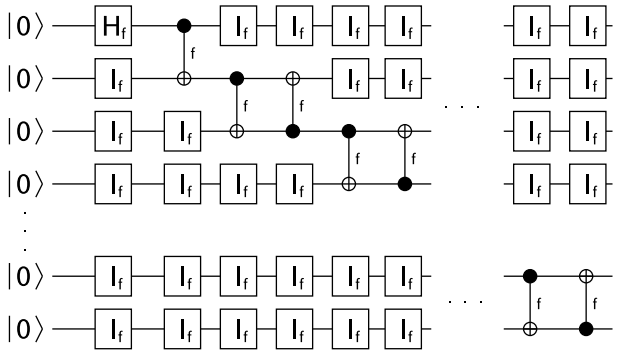


Figure 6.6: The remote EPR pair generation circuit with gate and systematic errors (see Figure 6.5 for the error-free version). A different randomly generated ε error parameter may be used for each gate. The total number of gates in the circuit is $(n-1)^2+n$.

Modeling random continuous gate error in the remote EPR pair generator (Figure 6.5) can be achieved by replacing the Hadamard gate with $H_f(\pi/4,\varepsilon)$, and a CNOT gate with $CNOT_f(0,\pi/2,\varepsilon_i,\varepsilon_{i+1})$. Reversing the tensor products of $CNOT_f$ generates a faulty CNOT gate with reversed control and target qubits.

In addition to gate errors, some quantum computing technologies are vulnerable to another type of error called systematic error or nonresonant effects [9, 11]. In the presence of systematic error, applying a gate to qubits i and $i+1$ *has a small effect on all other qubits*. To apply these small error effects, when any gate G_f is applied to one or more qubits, faulty identity gates of the form $I_i(0,\varepsilon_i)$ are applied to all other qubits not acted upon by G_f. This is a consistent model since in the error-free case, identity gates are implicitly present when a qubit is not acted upon by a gate.

The new form of the remote EPR pair generation circuit which includes gate and systematic error is shown in Figure 6.6. By adding in the faulty identity gates, the total number of gates for the faulty circuit is $(n-1)^2+n$. Assuming the worst-case conditions for

QuIDD-based simulation, a different randomly generated ε should be used in each faulty gate including the systematic error identity gates, which reduces the number of repeated values that the QuIDDs compress. Other error models may cause the number of different ε values to grow more rapidly, but such models are no harder to simulate with QuIDDs than the case considered. Since 1000 qubits are easily simulated for this benchmark, other error conditions can be simulated efficiently.

Jozsa describes a simple set of circuit reduction rules which may be applied to this circuit to analyze the difficulty of simulation for several different techniques [38]. Essentially, all 1-qubit gates may be merged via matrix multiplication into neighboring 2-qubit gates, and all neighboring 2-qubit gates applied to the same qubits may also be merged. If this reduction is performed on the faulty circuit shown in Figure 6.6, the resulting circuit contains only 2-qubit gates which are applied in a cascade fashion as shown in Figure 6.7.

It is clear from the reduced circuit that after each 2-qubit gate is applied to qubits i and $i+1$, qubit i is no longer affects the computation and may be removed via the partial trace (with the exception of the first and last qubits). In fact, as the EPR pair propagates down to the last qubit, each intermediate qubit may be dynamically tensored in with the previous EPR pair to create the EPR triple. After applying the current 2-qubit gate, the middle qubit in the triple may be traced out. Using the dynamic tensor product and partial tracing technique discussed earlier, the space complexity of simulating this circuit is reduced to $O(1)$ and the time complexity is reduced to $O(n)$. Given that random, continuous errors normally cause QuIDDs to blow up exponentially in size, this optimization offers an asymptotic improvement as verified experimentally in the next subsection.

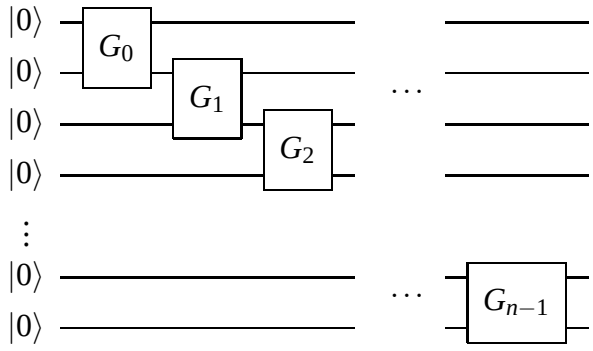


Figure 6.7: Reduced version of the faulty, remote EPR pair generation circuit.

We consider "collective dephasing" decoherence, which is known to be a major source of decoherence error in the ion trap implementation [40]. This type of decoherence can be modeled as phase dampening and can be simulated with a single "environment qubit" which couples to each data qubit through a controlled-Y gate as shown in Figure 6.8 [51]. The angle parameter to the controlled-Y gate is a decoherence angle, where angles closer to π model a more rapid decoherence process [51]. For simplicity, our experiments assume the measurement outcome of the environment qubit is always $|0\rangle$.[3] From the perspective of simulation, the key fact to note is that since the environment qubit is measured each time phase dampening is applied, it assumes a classical state and is no longer coupled to the data qubit.[4] Thus, decoherence only adds $O(1)$ runtime overhead using the dynamic tensoring and partial tracing technique for this circuit because the environment qubit can be removed via the partial trace.

It is important to note that p-blocked simulation, Vidal's slightly entangled technique,

[3]Though an outcome of $|1\rangle$ would immediately force the data qubit to $|1\rangle$, the fact that the environment is represented by a qubit is a simplification, and an outcome of $|1\rangle$ is not well-defined. The state of the environment should typically be the "ground state" which in the qubit model is $|0\rangle$.

[4]One-way computation heavily relies on single-qubit measurements [16].

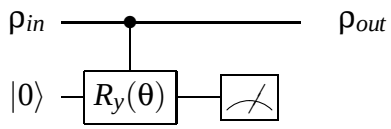


Figure 6.8: Phase-damping decoherence model involving an environment qubit.

and tensor networks can all simulate this circuit efficiently in the presence of these errors as well. However, runtime overhead can be incurred due to swaps in Vidal's technique. These implications are discussed more in Chapter VII.

6.2.5 Empirical Results

We used the dynamic tensoring and partial tracing technique in QuIDDPro to efficiently calculate the measurement outcome probabilities of all qubits in a faulty remote EPR pair generation circuit. As noted in Subsection 6.2.4, a reasonable rotation error range for faulty gates is 10^{-5} to 10^{-7} [29, 9, 11]. As a result, we consider three different cases in which random rotation errors are selected from normal distributions with ranges $\pm 10^{-5}$, $\pm 10^{-6}$, and $\pm 10^{-7}$, respectively. For each error distribution, we consider the remote EPR pair generation circuit with gate error alone and with gate and systematic errors together (decoherence error is considered later). In each case, the probability of error is calculated as $1 - P(|00\ldots0\rangle) - P(|10\ldots1\rangle)$, because in the absence of errors the probabilities of obtaining these outcomes should sum to 1. Also, since each gate is given its own randomly generated rotation error parameter, we compute the average of 100 different runs per error distribution.

Figures 6.9a-6.9c depict the the probability of error due to gate error only as a function

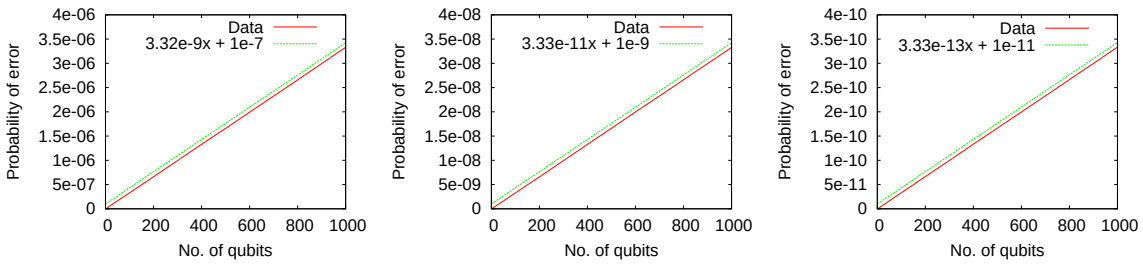


Figure 6.9: Probability of error in the remote EPR pair generation circuit due to gate error only, as a function of the number of qubits. The rotation errors are randomly selected for each gate from normal distributions ranging from (a) $\pm 10^{-5}$, (b) $\pm 10^{-6}$, and (c) $\pm 10^{-7}$. The average of 100 runs is used for each distribution.

of the number of qubits in the remote EPR pair generation circuit. The data indicates that the probability of error increases linearly with the number of qubits. Figures 6.10a-6.10c depict the probability of error due to gate error and systematic error as a function of the number of qubits. This data, however, indicates that in the presence of gate and systematic error, the probability of error increases *quadratically* with the number of qubits. This asymptotic difference between the two cases as a function of the number of qubits is not too surprising given that the number of faulty gates which must be simulated when modeling systematic error is quadratic in the number of qubits.

To model the growth of error as a function of the number of gates, the circuit size is fixed at 1000 qubits. This provides a good growth trend because the application of each pair of CNOT gates in sequence essentially models a remote EPR pair generation circuit with one more qubit. In other words, applying CNOT gates up to and including qubits i and $i+1$ is equivalent to simulating a remote entanglement circuit with only $i+1$ qubits. Thus, the error trend for a 1000-qubit remote EPR pair generation circuit as a function of the number of gates represents the trend for all remote EPR pair generation circuits of

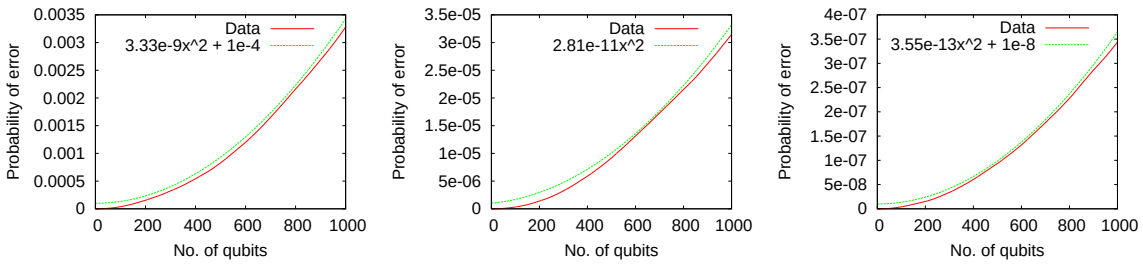


Figure 6.10: Probability of error in the remote EPR pair generation circuit, due to gate error and systematic error, as a function of the number of qubits. The rotation errors are randomly selected for each gate from normal distributions ranging from (a) $\pm 10^{-5}$, (b) $\pm 10^{-6}$, and (c) $\pm 10^{-7}$. The average of 100 runs is used for each distribution.

size up to and including 1000 qubits. Figures 6.11a-6.11c depict the probability of error in a 1000 qubit circuit in the presence of gate error only as a function of the number of gates. Note that the faulty identity gates are not counted since systematic error is not an actual gate applied by the implementer of the quantum computer. The data indicates that the probability of error increases linearly with the number of gates. Figures 6.12a-6.12c depict the probability of error in the presence of gate and systematic error. This data indicates that the growth of error increases quadratically with the number of gates. The similarity in growth trends as a function of the number of gates is not surprising since the number of gates applied to each qubit is a constant with respect to n.

In all cases, the magnitude of error is very small, even though the error in the presence of systematic error is several orders of magnitude larger than in the absence of systematic error. More importantly, the growth of error as functions of the number of qubits and gates is sub-exponential. As a result, since remote EPR pair generation is a key step in quantum teleportation, error correction aimed at gate errors and/or systematic error is probably not

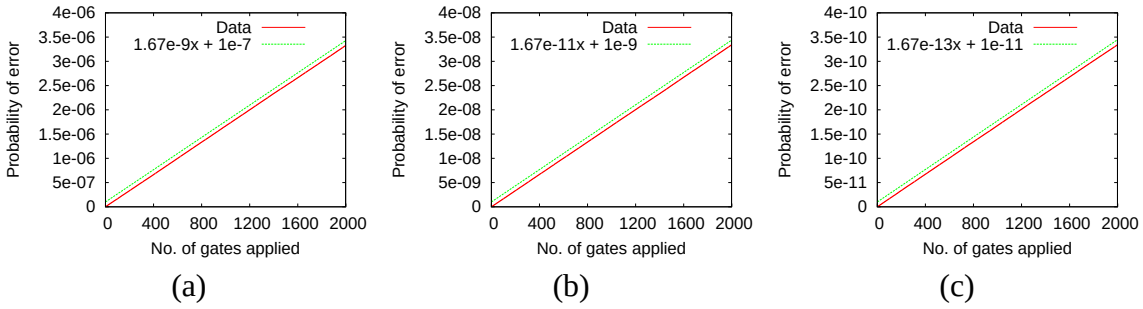


Figure 6.11: Probability of error in the remote EPR pair generation circuit due to gate error only, as a function of the number of gates. The rotation errors are randomly selected for each gate from normal distributions ranging from (a) $\pm 10^{-5}$, (b) $\pm 10^{-6}$, and (c) $\pm 10^{-7}$. The average of 100 runs is used for each distribution.

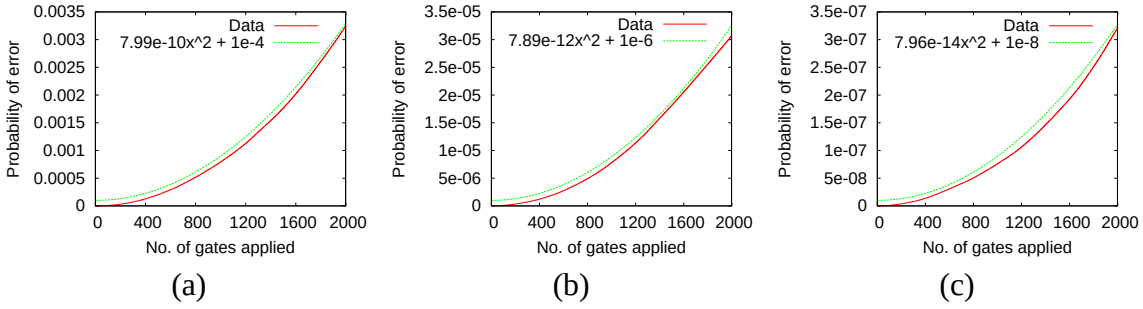


Figure 6.12: Probability of error in the remote EPR pair generation circuit due to gate error and systematic error, as a function of the number of gates. The rotation errors are randomly selected for each gate from normal distributions ranging from (a) $\pm 10^{-5}$, (b) $\pm 10^{-6}$, and (c) $\pm 10^{-7}$. The average of 100 runs is used for each distribution.

necessary for quantum teleportation of a qubit state.

We also simulated the circuit in the presence of collective-dephasing decoherence error modeled as phase dampening. Whereas gate and systematic errors tend to increase the probability of measuring incorrect outcomes, phase dampening also skews the probability distribution of measuring the correct outcomes $|00\ldots0\rangle$ or $|10\ldots1\rangle$ (i.e. the probabilities of measuring one correct state instead of the other are not equal). Thus, a better metric for these experiments is the fidelity of the faulty state σ as compared to the correct state ρ.

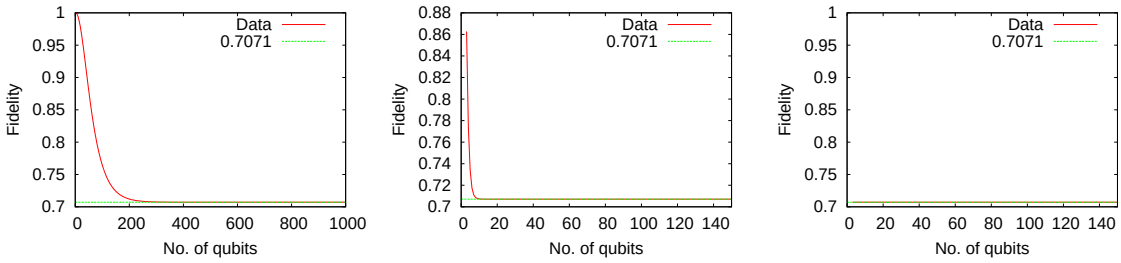


Figure 6.13: State fidelity in the remote EPR pair generation circuit with decoherence as a function of the number of qubits. The decoherence angles used are (a) 0.2, (b) 1.2, and (c) 3.0. Results are only shown for up to 140 qubits for (b) and (c) since the fidelity drops to approximately $1/\sqrt{2}$ quickly.

For density matrices, this is expressed as, $F(\rho,\sigma) = tr\sqrt{\rho^{1/2}\sigma\rho^{1/2}}$, where $F(\rho,\sigma)$ ranges between 0 (the states are completely different) and 1 (the states are equal) [51].

The first set of experiments simulate phase dampening alone with three different decoherence angles, 0.2, 1.2, and 3.0. The results are shown in Figures 6.13a-c, and they confirm that the fidelity drops much more quickly for decoherence angles closer to π. The second set of experiments simulate phase dampening with bang-bang error correction [87, 46, 86, 39, 57]. There are many ways to define the bang-bang corrective operators. In these experiments, the "universal decoupling" sequence is used, which alternates between the Pauli X and Z operators after every gate is applied [86, 39]. Compared to corrective operators that involve negations of the decoherence operator itself [87], this choice is arguably more realistic and useful to experimental physicists since it requires no knowledge of the Hamiltonian representing the underlying decoherence process. As shown in Figures 6.14a and 6.14b, this set of bang-bang operators is extremely effective for this particular circuit. Unlike the previous results, the fidelity never reaches 0. However, Figure 6.14c shows that the extremely rapid decoherence process modeled by decoherence angle 3.0 is

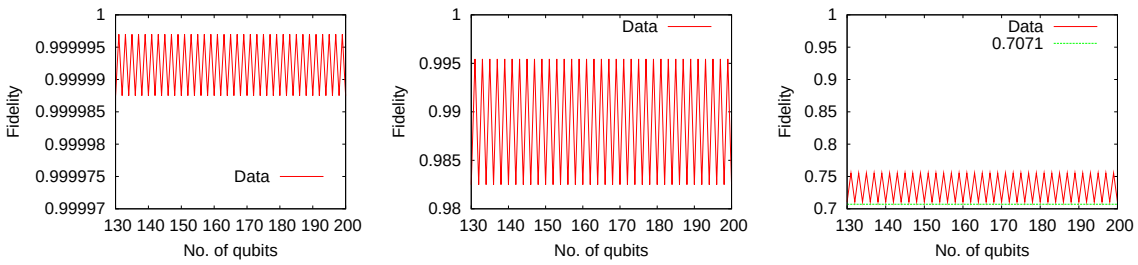


Figure 6.14: State fidelity in the remote EPR pair generation circuit with decoherence as a function of the number of qubits. Bang-bang pulses from the universal decoupling sequence are used to correct the state after every gate is applied. The decoherence angles used are (a) 0.2, (b) 1.2, and (c) 3.0. Results are given from 130 to 200 qubits so that the periodic nature of the data is easily viewed. The trends continue through 1000 qubits.

not effectively dealt with by this choice of operators. Since this angle is so close to π, there may be no practical way to cope with such a rapid decoherence process using bang-bang operations.

An open question is how effective bang-bang operators are in the presence of gate error due to imprecision [88, 39]. Figures 6.15a-c provide data from the last set of experiments which model phase dampening, the bang-bang operators used in the previous experiments, and a gate error range of $\pm 10^{-5}$ (the worst-case range). Interestingly enough, the bang-bang operators are indeed able to cope with decoherence angles 0.2 and 1.2 as before, suggesting that gate imprecision may not be a significant problem for bang-bang error correction.

6.3 Summary

This chapter discussed two important language-level techniques for state-vector and density-matrix simulation. In the case of state vector simulation, the simulator automati-

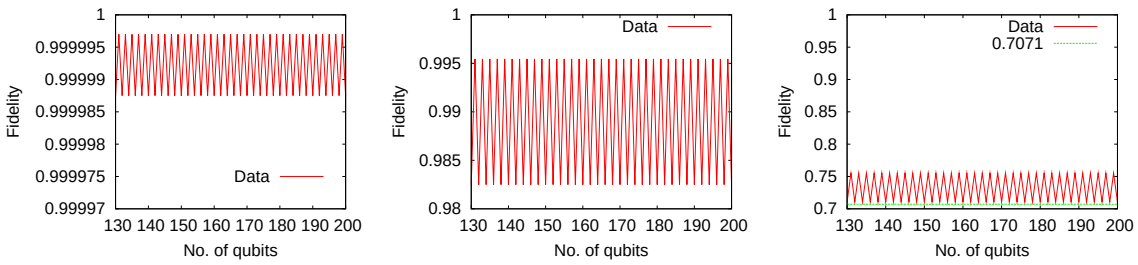


Figure 6.15: State fidelity in the remote EPR pair generation circuit with decoherence as a function of the number of qubits. Faulty bang-bang pulses from the universal decoupling sequence with an error range $\pm 10^{-5}$ are used to correct the state after every gate is applied. The decoherence angles used are (a) 0.2, (b) 1.2, and (c) 3.0. Results are given from 130 to 200 qubits so that the periodic nature of the data is easily viewed. The trends continue through 1000 qubits.

cally detects when to apply specialized gate-specific algorithms depending on the expressions used. This demonstrates that QuIDDs are competitive with stabilizer simulation for several large benchmarks, including one benchmark that contains non-stabilizer Toffoli gates. In the case of density matrix simulation, particular language features are utilized to introduce qubits only when they are needed and remove them when they no longer affect the qubits of interest. This approach is used to characterize gate, systematic, and decoherence errors as well as bang-bang error correction in the remote EPR pair generation circuit.

CHAPTER VII

Conclusions

In the worst case, quantum circuit simulation requires runtime and memory resources that grow exponentially with the number of qubits simulated. Quantum circuits are also significantly more complicated than classical digital logic circuits, and their unique properties are difficult to capture using traditional CAD techniques. One of the most important of these properties is the fragile nature of quantum information. Quantum states are often damaged over time by several types of gate-specific and environmental errors, which experimental physicists find difficult to characterize. Additionally, the notion of equivalence, while trivial in the classical case, takes on a surprisingly rich set of interpretations in the quantum case, offering several computational challenges of varying complexity. As a result, useful quantum CAD tools must incorporate special models and efficient, classical simulation techniques to overcome these obstacles for classes of circuits with practical value.

Our work centers around developing such techniques and implementing them in a unified framework. The algorithms, data structures, and simulation package we developed provide an efficient testbed for analyzing quantum circuit properties via simulation. In the

remaining sections, a detailed discussion of our contributions is offered, followed by some final perspectives and a discussion of future applications.

7.1 Summary of Contributions

In this thesis, we have evaluated a number of simulation techniques, contributed a new and efficient technique of our own, and developed a comprehensive software tool based on this technique. Our contributions directly facilitate further further analysis of quantum speed-ups, exploitable structure in quantum information, error characterization, language development, and synthesis, among others. A summary of the major contributions of this dissertation is as follows:

- Development of the QuIDD data structure and QuIDD-based algorithms for general-purpose quantum circuit simulation. The algorithms cover all of the key simulation operations.

- Implementation and evaluation of the QuIDDPro simulator which supports general simulation of both state vector and density matrix representations of quantum circuits and offers approximately 100 functions to provide a wide range of simulation applications to quantum circuit CAD.

- Fast, memory-efficient QuIDD simulation of numerous benchmark circuits, demonstrating the practical value of our technique. We also formally describe a class of states and operators that require $O(n)$ or smaller time and memory resources when simulated with QuIDDs.

- Algorithms and other techniques enabling QuIDDs to simulate circuits using the density matrix model, including the ability to trace over qubits which represent environmental interference or other effects.

- Fast algorithms that exploit QuIDD properties to perform equivalence checking up to global and relative phase of both states and operators.

- Specialized gate algorithms and pre-processing algorithms that dramatically enhance the performance of applying controlled- and 1-qubit gates to QuIDDs. This allows QuIDDs to enjoy some of the advantages of the stabilizer formalism on large circuit benchmarks without the severe limitation on allowed gate types.

- Algorithms which enable QuIDD characterization of several common forms of random, continuous errors as well as error correction in a case study circuit that creates remote EPR pairs via faulty nearest-neighbor interactions.

In Chapter II, we surveyed the major quantum circuit simulation techniques. Each technique exploits some form of structure in quantum information. In addition to these techniques, we described our QuIDD data structure and QuIDD algorithms in Chapter III. This chapter focuses on the state vector model of quantum computation and describes a class of states and operators which can be represented with $O(n)$ size complexity, including instances of Grover's search algorithm depending on the search criteria encoded by the oracle. The results build the foundations for addressing the goals of quantum circuit simulation, which are characterizing the effect of various errors in practical quantum circuits, testing multi-qubit error correction techniques to cope with such errors, verifying

the correctness of synthesized quantum circuits, and exploring the boundaries between the quantum and classical computational models. They also illuminate the boundaries between the quantum and classical computational models.

QuIDD algorithms are introduced in Chapter IV to model density matrices, which provides specific tools to address the first two goals of quantum circuit simulation, namely characterizing physical error effects and evaluating error correction techniques. The QuIDD algorithms implement key density matrix operations such as the partial trace and outer product. Benchmarks which include errors, error correction, reversible logic, quantum communication, and quantum search were used to demonstrate that QuIDDs offer dramatic practical improvements over NIST's QCSim package. Although QuIDD representations of density matrices have exponential size complexity in general, the benchmark results demonstrate that the class of states and operators which QuIDDs represent efficiently includes important applications.

Classical CAD tools are frequently used to reduce the size of logic circuits while preserving equivalent functionality. In this context, a major use of simulation is to verify functional equivalence. Although the canonicity of ordered decision diagrams facilitates exact equivalence checking with QuIDDs in only $O(1)$ time, equivalence checking up to global and relative phases does not enjoy this property. Therefore we provided other equivalence-checking algorithms in Chapter V to check for looser equivalence relations. We showed that global- and relative-phase equivalence among states and operators may be checked efficiently for a number of benchmarks ranging from Hamiltonian simulation to quantum number factoring.

Lastly, in Chapter VI, we discussed algorithms which speed-up key simulation operations in certain situations. In the case of state vectors, pre-processing can be used to distinguish controlled- and 1-qubit gates from arbitrary gates. We presented several specialized algorithms which are utilized automatically by the simulator instead of QuIDD matrix multiplication to optimize simulation of such gates. These optimizations make QuIDDs competitive with specialized simulation methods such as the stabilizer formalism for a variety of large stabilizer circuit benchmarks. Additionally, we showed how dynamic tensor products and tracing over unentangled qubits enables QuIDDs to simulate different forms of continuous, random error including decoherence. Relevant language features allow simulation of such errors in the n-qubit remote EPR pair generation circuit using $O(1)$ memory and $O(n)$ time, which is an asymptotic improvement over using an n-qubit QuIDD state vector and no tracing. Furthermore, decoherence error and "bang-bang" error correction were simulated in this circuit with only $O(1)$ time and memory overhead versus the $O(n)$ time overhead required by Vidal's technique due to qubit swapping. Simulation of these errors in addition to error correction addresses the first two goals of quantum circuit simulation.

In the next section we discuss how the QuIDD data structure, algorithms, and language properties relate to some of the other major simulation techniques surveyed earlier. We also provide several final perspectives on quantum CAD and ideas for future developments in the field.

7.2 Closing Remarks and Future Directions

Although all of the techniques discussed efficiently simulate different classes of quantum circuits depending on various properties, it is still unclear how much overlap exists among these techniques for practical simulation applications. For example, the remote EPR pair generation circuit analyzed in Chapter VI can also be simulated by Vidal's technique or tensor networks. However, decoherence errors can induce $O(n)$ swaps when using Vidal's method. This overhead appears avoidable by dynamically concatenating single-qubit tensors to Vidal's tensor decomposition, similar to what is done with QuIDDs and dynamic tensor products in Chapter VI. The concatenation should be straightforward when introducing a new qubit in the ground state since there is no entanglement between the current state and the new qubit ($\chi = 1$). The partial trace would also be required to remove the environment qubit after decoherence and measurement are applied.

This example illustrates several key points about practical quantum circuit simulation. First, regardless of the back-end simulation technique, the front-end language and supporting functionality are important. Without the power to express certain simulation optimizations, such as dynamic tensor products and partial tracing, or specialized gates, a great deal of computational resources may be wasted. In some cases, such unnecessary runtime and memory overhead may grow asymptotically.

Second, formal descriptions of *how* the various classes of efficiently simulatable quantum circuits overlap would provide a powerful tool for further work in theoretical computation. For instance, new developments are appearing which combine some of the simulation techniques discussed in this dissertation. Shi, Duan and Vidal offer such an approach

by replacing the tensors in Vidal's decomposition with tensor networks [64]. Whereas it was shown that Vidal's technique alone efficiently simulates one-dimensional quantum many-body systems [85] and tensor networks alone efficiently simulate tensor networks with low tree width, the hybrid approach efficiently simulates quantum many-body systems of arbitrary dimension so long as their tensor network representation has low tree width [64]. In addition to enabling hybrid approaches, formal descriptions of the overlap among techniques could make the incorporation of powerful, classical data structures more transparent. Anders and Briegel's replacement of the bit tables with a graph-based data structure in the stabilizer formalism reduced the complexity of simulating that particular class of quantum circuits [3]. QuIDDs too can be viewed as an analogous replacement for explicit matrices and vectors. Such data structures draw from many seemingly disjoint areas of computer science and engineering, ranging from theoretical algorithmic analysis to heuristic CAD for classical digital logic design. It is likely that other classical data structures, algorithms, and heuristics exist which will further benefit quantum circuit CAD.

Lastly, studying more benchmark circuits of interest with various types of physically realistic errors is crucial to expanding the practical value of the various simulation techniques. Although there is great theoretical value in identifying the classes of circuits that may be simulated efficiently by all of the different techniques, experimental physicists, like electrical engineers who design modern classical processors, have very practical requirements for specific applications. For quantum circuit CAD to find use as a practical tool, theoretical simulation results should be treated as a foundation on which to build robust, efficient software packages. QuIDDPro is an "end-to-end" project which started

with our theoretical contributions to simulation and culminated in a rich software package aimed at providing physicists and many other researchers in the field with a useful tool.

As illustrated by our discussions of other simulation techniques, it is clear that there are many solid foundations on which to expand quantum circuit CAD development. We hope that QuIDDPro will serve as a helpful example of how to pursue such development. The continued efforts of many researchers in the emerging field of quantum computation combined with decades of experience accumulated in classical circuit CAD will undoubtedly produce even more powerful tools for quantum circuit design.

APPENDICES

APPENDIX A

A Characterization of Persistent Sets

The following sequence of lemmas published in [80] leads to a complete characterization of persistent sets from Definition 3.8. This definition considers finite non-empty sets of complex numbers Γ_1 and Γ_2, and denotes their *all-pairs product* as $\{xy \mid x \in \Gamma_1,\ y \in \Gamma_2\}$. One can verify that this operation is associative, and therefore the set Γ^n of *all n-element products* is well defined for $n > 0$. We then call a finite non-empty set $\Gamma \subset \mathbb{C}$ *persistent* iff the size of Γ^n is constant for all $n > 0$. We start by observing that adding 0 to, or removing 0 from, a set does not affect its persistence.

Lemma A.1 *All elements of a persistent set* Γ *that does not contain 0 must have the same magnitude.*

Proof. For Γ to be persistent, the set of magnitudes of elements from Γ must also be persistent. Therefore, it suffices to show that each persistent set of positive real numbers contains no more than one element. Assume, by way of contradiction, that such a persistent set exists with at least two elements r and s. Then among n-element products from Γ, we find all numbers of the form $r^{n-k}s^k$ for $k = 0..n$. If we order r and s so that $r < s$, then it becomes clear that the products are all different because $r^{n-k+1}s^{k-1} < r^{n-k}s^k$. $\square$

Lemma A.2 *All persistent sets without 0 are of the form $c\Gamma'$, where $c \neq 0$ and Γ' is a finite persistent subset of the unit circle in the complex plane $\mathbb{C}$, containing 1 and closed under multiplication. Vice versa, for all such sets Γ' and $c \neq 0$, $c\Gamma'$ is persistent.*

Proof. Take a persistent set Γ that does not contain 0, pick an element $z \in \Gamma$ and define $\Gamma' = \Gamma/z$, which is persistent by construction. Γ' is a subset of the unit circle because all numbers in Γ have the same magnitude. Due to the fact that $z/z = 1 \in \Gamma'$, the set of n-element products contains every element of Γ'. Should the product of two elements of Γ' fall beyond the set, Γ' cannot be persistent. □

Lemma A.3 *A finite persistent subset $\Gamma' \ni 1$ of the unit circle that is closed under multiplication must be of the form $\mathbb{U}_n$ (roots of unity of degree n).*

Proof. If $\Gamma' = \{1\}$, then $n = 1$, and we are done. Otherwise consider an arbitrary element $z \neq 1$ of Γ' and observe that all powers of z must also be in Γ'. Since Γ' is finite, $z^m = z^k$ for some $m \neq k$, hence $z^{m-k} = 1$, and z is a root of unity. Therefore Γ' is closed under inversion, and forms a group. It follows from group theory, that a finite subgroup of $\mathbb{C}$ is necessarily of the form $\mathbb{U}_n$ for some n. □

Theorem A.4 *Persistent sets are either of the form $c\mathbb{U}_n$ for some $c \neq 0$ and n, or $\{0\} \cup c\mathbb{U}_n$.*

APPENDIX B

QuIDDPro Simulator

QuIDDPro is a quantum circuit simulator we have developed around our QuIDD data structure and QuIDD-based algorithms. It provides numerous built-in functions and language features which make QuIDDs transparent and easy to use. This appendix provides a brief overview of how to run the simulator as well as a language reference.

B.1 Running the Simulator

The QuIDDPro simulator can be run in two modes, namely batch mode and interactive mode. In batch mode, the user supplies the simulator with an ASCII text file containing the script code to be executed. The text file can be provided as an argument in the command line to the simulator executable or redirected to standard input as in the following examples:

File “my_code.qpro” passed as an argument:

```
% ./qp my_code.qpro
```

File “my_code.qpro” redirected to standard input:

```
% ./qp < my_code.qpro
```

Note that although the examples use a ".qpro" extension in the filenames, any valid filename will do.

Interactive mode is triggered when the simulator executable is given no arguments at the command line. In this mode, the simulator will be started and produce a prompt to await input from the user as shown in the next example:

```
% ./qp
QuIDDPro>
```

Similar to MATLAB, valid lines of code may be typed at the prompt and executed when the return or enter key is pressed (i.e. when a newline is given as input). The command "quit" can be issued to exit the simulator. Also, multiple expressions may be placed in a single line by separating each expression by one or more semicolons. An example of this method of input is as follows:

```
QuIDDPro> a = pi/3; r_op = [cos(a/2) -i*sin(a/2); -i*sin(a/2) cos(a/2)]
r_op =
 0.866025 0-0.5i
 0-0.5i 0.866025
```

In this example, a 1-qubit rotational X operator matrix is created with the θ parameter $\pi/3$. Notice that only the value of the variable "r_op" is printed out. In general, the value of the last expression is printed out for an input line containing multiple expressions separated by semicolons. However, the other expressions are still computed. In this example, for instance, the variable "a" will contain the value $pi/3$, even though this result is not printed out. This is clearly true since the definition of "r_op" depends on the value of "a." In

addition to providing the means to place multiple expressions on the same line, semicolons can be used more generally to suppress output to the screen. If screen output for any particular expression is not desired, simply place a semicolon at the end of the expression to compute it silently. MATLAB behaves in the same fashion.

QuIDDPro contains a number of built-in functions and predefined variables. A listing of such functions and variables can be found in Section B.3. Notice that in the last example, the predefined variables "pi" and "i" are used. "pi" contains the value π (to a large number of digits), while "i" contains the value $0 + i$. Predefined variables can be overwritten by the user. In addition to the predefined variables just mentioned, the built-in functions "cos" and "sin" were also used in the last example. To demonstrate the use of built-in functions further, consider the next example:

```
QuIDDPro> r_op = rx(pi/3, 1)

r_op =

 0.866025 0-0.5i

 0-0.5i 0.866025
```

In this example, the built-in function "rx" is used to create the same matrix that was created in the previous example, namely the 1-qubit rotational X operator. QuIDDPro provides a number of such functions to create commonly used operators. See Section B.3 for more details.

Although interactive mode is useful for quick calculations, it may not be preferable for non-trivial pieces of code that are reused many times. Thus, batch mode is highly recommended for most contexts. In the next example, we demonstrate how to use QuIDD-

Pro to simulate a quantum circuit in batch mode. The code shown here can be placed into a file for execution at any time. In fact, this particular example and others can be found in the examples/ directory.

Consider the canonical decomposition of a two-qubit unitary operator U described in [21]. U can be expressed as:

$U = (A_1 \otimes B_1) e^{i(\theta_x X \otimes X + \theta_y Y \otimes Y + \theta_z Z \otimes Z)} (A_2 \otimes B_2)$

subject to the constraint that $\frac{\pi}{4} \geq \theta_x \geq \theta_y \geq |\theta_z|$ and A_1, A_2, B_1, and B_2 are one-qubit unitary operators.

Suppose we wish to simulate a quantum circuit in which some two-qubit unitary operator U is to be applied to two qubits in the density matrix state $|10\rangle\langle 10|$. Further suppose that U must be computed given the canonical decomposition parameters $\theta_x = 0.702$, $\theta_y = 0.54$, and $\theta_z = 0.2346$. Additionally, we are given that A_1 is a one-qubit Hadamard operator, A_2 is X, B_1 is I, and B_2 is Y. This can be implemented with the following code (from examples/misc/two_q_canonical.qpro):

```
theta_x = 0.702;

theta_y = 0.54;

theta_z = 0.2346;

A1 = hadamard(1);

A2 = sigma_x(1);

B1 = identity(1);

B2 = sigma_y(1);
```

Next, U can be computed with the code:

```
Xpart = theta_x*kron(sigma_x(1), sigma_x(1));

Ypart = theta_y*kron(sigma_y(1), sigma_y(1));

Zpart = theta_z*kron(sigma_z(1), sigma_z(1));

U = kron(A1, B1)*expm(i*(Xpart + Ypart + Zpart))*kron(A2, B2)
```

U is then applied to the density matrix state $|10\rangle\langle 10|$ with the code:

```
state = cb(''10'');

final_state = U*(state*state')*U'
```

Deterministic measurement can be performed to eliminate the correlations associated with each qubit:

```
q_index = 1;

while (q_index < 3)

final_state = measure(q_index, final_state);

q_index = q_index + 1;

end

measured_state = final_state
```

U can also be applied very easily to the state vector representation of the state if it is preferred to the density matrix representation. In addition, the probability of measuring a 1 or 0 for any qubit in the state vector can be computed using other measurement functions:

```
final_state_v = U*state

p0_qubit1 = measure_sv0(1, final_state_v)

p1_qubit1 = measure_sv1(1, final_state_v)

p0_qubit2 = measure_sv0(2, final_state_v)
```

```
p1_qubit2 = measure_sv1(2, final_state_v)
```

Probabilistic measurement can also be performed on both density matrices and state vectors. See `pmeasure` and `pmeasure_sv` in Section B.3 for more details.

Upon execution of the above script, the output is:

```
U =

-0.110927-0.0265116i  -0.0530448-0.222078i -0.650863+0.15556i 0.162218-0.678733i

-0.162218+0.678733i   0.650863-0.15556i   0.0530448+0.222078i 0.110927+0.0265116i

-0.110927-0.0265116i  0.0530448+0.222078i 0.650863-0.15556i   0.162218-0.678733i

0.162218-0.678733i    0.650863-0.15556i   0.0530448+0.222078i -0.110927-0.0265116i

final_state =

0.447822   2.15483e-05+0.152794i -0.447822    2.15483e-05+0.152794i

2.15483e-05-0.152794i 0.0521324  -2.15483e-05+0.152794i   0.0521324

-0.447822 -2.15483e-05-0.152794i  0.447822   -2.15483e-05-0.152794i

2.15483e-05-0.152794i 0.0521324  -2.15483e-05+0.152794i   0.0521324

measured_state =
 0.447822      0          0          0
    0      0.0521324      0          0
    0          0      0.447822       0
    0          0          0      0.0521324


final_state_v =

-0.650863+0.15556i

0.0530448+0.222078i

0.650863-0.15556i

0.0530448+0.222078i
```

```
p0_qubit1 =

0.499955

p1_qubit1 =

0.499955

p0_qubit2 =

0.895644

p1_qubit2 =

0.104265
```

Although the examples in this section demonstrate scripts that use small numbers of qubits, the real power of QuIDDPro lies in simulating quantum-mechanical systems with many quantum states (usually 10 or more). See steaneX.qpro, steaneZ.qpro, and large_h.qpro in the examples/ directory for examples of such systems. large_h.qpro, for instance, applies a 50 qubit Hadamard operator to a density matrix of 50 qubits. steaneX.qpro and steaneZ.qpro demonstrate error correction in quantum circuits of 12 and 13 qubits, respectively. On a single-processor workstation, each of these scripts requires less than 5 seconds to run and less than 0.5 MB of peak memory usage.

B.2 Functions and Code in Multiple Files

QuIDDPro supports user-defined functions via the "m-file" model commonly used in MATLAB. Specifically, a function call to a user-defined function may appear anywhere as long as the function body is contained in a separate file in the working directory. The name of the file containing the function body must be the same as the function name with

".qpro" or ".qp" appended. To illustrate, consider the following script which uses an oracle function to implement a simple instance of Grover's algorithm shown on page 256 of [51]. Notice that Dirac-style syntax maybe used for state vector QuIDDs.

(*examples/functions/simple_grover.qpro*)

```
|state:> = cb(``001'');

|state:> = hadamard(3)*|state:>;

|state:> = oracle(|state:>);

|state:> = cu_gate(hadamard(1), ``xxi'')*|state:>;

|state:> = cu_gate(sigma_x(1), ``xxi'')*|state:>;

|state:> = cu_gate(hadamard(1), ``ixi'')*|state:>;

|state:> = cu_gate(sigma_x(1), ``cxi'')*|state:>;

|state:> = cu_gate(hadamard(1), ``ixi'')*|state:>;

|state:> = cu_gate(sigma_x(1), ``xxi'')*|state:>;

|state:> = hadamard(3)*|state:>
```

(*examples/functions/oracle.qpro*)

```
function |new_state:> = oracle(curr_state)

   |new_state:> = cu_gate(sigma_x(1), ``ccx'')*|curr_state:>;
```

The user-defined function is "oracle" with its function body defined in the file "oracle.qpro." The other functions used are part of the QuIDDPro language (see Section B.3 for more details). Notice that in this particular example, the QuIDD "state" is passed as a function argument. In QuIDDPro, a QuIDD function argument only requires $O(1)$ memory usage because a pointer to the head of the QuIDD is passed to a function rather than

the entire QuIDD. The same holds true for returning QuIDDs from a function. Thus, passing QuIDD arguments and return values is extremely efficient. In general, a user-defined function can contain any number of parameters which can be any combination of QuIDDs or complex numbers. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used).

Unlike MATLAB, QuIDDPro functions must have only one return variable (a function that returns nothing is also not allowed). If the function is intended to return no values, such as a diagnostic printing function, then a dummy variable can be used for the return variable. The return variable need not be used in the function body, and when this occurs, it is automatically assigned a value of 0. A semicolon can be appended to the function call to suppress the output of the 0 value. When multiple return values are desired, they can be stored together in a matrix. Thus, requiring a single return variable does not actually restrict the number of values that can be returned.

Like MATLAB and other languages, variables declared locally in a function body exist in their own scope. In other words, variables declared in a function body are undefined upon leaving the function body. By the same token, such variables do not overwrite the values of variables with the same name declared outside the function body.

In addition to functions, QuIDDPro supports the *run* command. Like its MATLAB counterpart, this command runs script code contained in another file. In the following example, the same circuit as before is simulated, but this time the run command is used instead of a user-defined function.

(*examples/run/simple_grover.qpro*)

```
run ‘‘oracle_def.qpro’’

state = cb(‘‘001’’);

state = hadamard(3)*state;

state = oracle*state;

state = cu_gate(hadamard(1), ‘‘xxi’’)*state;

state = cu_gate(sigma_x(1), ‘‘xxi’’)*state;

state = cu_gate(hadamard(1), ‘‘ixi’’)*state;

state = cu_gate(sigma_x(1), ‘‘cxi’’)*state;

state = cu_gate(hadamard(1), ‘‘ixi’’)*state;

state = cu_gate(sigma_x(1), ‘‘xxi’’)*state;

state = hadamard(3)*state
```

(*examples/run/oracle_def.qpro*)

```
oracle = cu_gate(sigma_x(1), ‘‘ccx’’);
```

Notice that the run command does not introduce a new scope. All variables declared in a run file exist in the current scope. As a result, the run command is ideal for declaring variables which can be re-used in multiple projects. Also, there is no constraint on where a run command may appear other than that it may not be placed within an explicit matrix.

B.3 Language Reference

This section provides a reference for the QuIDDPro input language. Although the language is similar to MATLAB, there are many functions in QuIDDPro specific to quantum-mechanics which do not exist in MATLAB. There are also a large number of functions in MATLAB which are not supported by QuIDDPro. Additionally, some of the functions that have the same names as those in MATLAB have slightly different functionality from their MATLAB counterparts. New language features will be added in future versions of the QuIDDPro simulator, and we welcome user suggestions.

==	~=, !=	<	<=
>	>=	&&	||
+	-	*	/
=	'	(...)	∧

Operations

cutoff_val	i
output_prec	pi
qp_epsilon	r2
r3	

Predefined variables

[...]	;	a(n, k)
a($n_1, n_2, n_3, \ldots$)	else	elseif
function	if	run
tic	toc	while
for	end	

Language features

atan	cb	cnot	conj
cos	cps	cu_gate	dump_dot
echo	exp	expm	eye
fredkin	gen_amp_damp	hadamard	identity
kron	norm	measure	measure_sv
measure_sv0	measure_sv1	pmeasure	pmeasure_norm_sv
pmeasure_sv	proj0	proj1	projplus
ptrace	px, Px	py, Py	pz, Pz
quidd_info	rand	round	rx, Rx
ry, Ry	rz, Rz	sigma_x	sigma_y
sigma_z	sin	sqrt	swap
toffoli	zeros		

Built-in Functions

- [...] defines a matrix explicitly. Expressions are placed between the brackets. Elements in the same row are separated by whitespace (including newlines) or commas,

while rows are separated by one or more semicolons. The brackets can be nested within other brackets (matrices within matrices).

- # starts a one-line comment. Everything from the # symbol to the first newline is ignored. An alternative comment symbol is %.

- % starts a one-line comment. Everything from the % symbol to the first newline is ignored. An alternative comment symbol is #.

- ′ returns the complex-conjugate transpose of a matrix. For example, $[1\ 2;\ 3+2i\ 4]' \rightarrow [1\ 3-2i;\ 2\ 4]$

- == equality operation that returns 1 if the two expressions compared are equal; otherwise it returns 0. Comparison between matrices is supported. A complex number and a matrix are considered not equal unless the matrix has dimensions 1×1 and contains a number equal to the one being compared to.

- ~= inequality operation that performs the complement function of ==.

- != an alternative symbol for ~=.

- < less than operation. It returns 1 if the left-hand expression is less than the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.

- <= less than or equal operation. It returns 1 if the left-hand expression is less than or equal to the right-hand express; otherwise it returns 0. It can only be used to

compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.

- $>$ greater than operation. It returns 1 if the left-hand expression is greater than the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.

- $>=$ greater than or equal operation. It returns 1 if the left-hand expression is greater than or equal to the right-hand express; otherwise it returns 0. It can only be used to compare numbers. For numbers with nonzero imaginary components, only the real parts are compared.

- && logical AND connective. It returns 1 if both sides of the operator evaluate to 1; otherwise it returns 0. It can only be used to compare numbers with nonzero imaginary components.

- || logical OR connective. It returns 1 if either side of the operator evaluates to 1; otherwise it returns 0. It can only be used to compare numbers with nonzero imaginary components.

- $+$ addition operation. For complex numbers, it returns the sum of the numbers. For matrices, it returns the element-wise addition of both matrices (both matrices must have the same number of rows and columns). When a matrix is added to a complex number, the complex number is added to each element of the matrix as a scalar.

- $-$ subtraction operation. For complex numbers, it returns the difference of the numbers. For matrices, it returns the element-wise difference of both matrices (both matrices must have the same number of rows and columns). When a matrix is subtracted from a complex number or vice-versa, scalar subtraction is performed element-by-element. When there is no left-hand expression, it is treated as a unary minus applied to the right-hand side expression. Within a matrix definition, for example $[1-2]$, the minus sign is treated as a unary minus. However, in $[1-2]$ and $[1-2]$, the minus sign is treated as the binary minus expression. Parenthesis can be used to force the minus sign to be treated one way or the other.

- $*$ multiplication operation. For complex numbers, it returns the product of the numbers. For matrices, matrix multiplication is performed (as opposed to element-wise multiplication). Scalar multiplication is performed when a matrix and a complex number are multiplied together.

- $/$ division operation. For complex numbers, it returns the quotient. Unlike the C language, integer division is *not* performed if the operands are both integer values. Double floating point division is always performed. For matrices, element-wise division is performed (both matrices must have the same number of rows and columns). When a matrix is divided by a complex number, scalar division is performed. However, a complex number may not be divided by a matrix.

- $=$ assignment operation. It assigns the value of an expression (right-hand side) to a variable (left-hand side). The expression may result in either a complex number or a matrix. The left-hand side expression must be a variable name (it must start

with a letter and contain only alpha-numeric characters and optionally underscores). Variables can be assigned "on-the-fly." In other words, unlike languages like C/C++, variables are not declared nor typed in any way prior to their first assignment. However, a variable must be assigned a value before it can be used in an expression. Similar to languages such as C/C++, an assignment expression returns a value just like any other expression, namely the value that was assigned to the variable on the left-hand side. Therefore, statements such as $x = y = 3 + 4i$ are valid. In statements like these, if output is not suppressed, the value of the leftmost variable will be output to the screen. Although the other variables assigned values will not be output to the screen, they are still assigned their values. Another important note is that even though string literals appear as arguments in some functions, including *cu_gate* and *echo*, assignment of a string literal to a variable is not yet supported.

- $\wedge$ exponentiation operation for complex numbers. It returns the expression on the left-hand side of the $\wedge$ raised to the power of the expression on the right-hand side. For matrix exponentiation, see the `expm` function.

- $(\ldots)$ forces precedence for an expression as in any other programming language. An expression within the parentheses is evaluated before evaluating expressions outside of the parentheses.

- ; the semicolon suppresses output of an expression. For example, $x = 1$ stores the value of 1 in the variable x and output $x = 1$ to standard output, whereas $x = 1$; also stores the value of 1 in the variable x but would not output anything to standard output. When a semicolon appears in a matrix definition, it has a different meaning

entirely. Within a matrix definition, a semicolon denotes the end of a row.

- `a(`*n, k*`)` if `a` is a variable containing a matrix, then this expression returns the element indexed by the row index *n* and the column index *k*. Numbering of indices starts at 1. Unlike languages such as MATLAB, this expression may not be used to assign values to elements of a matrix. It may only be used to read a particular element from a matrix (e.g. `x = a(`*1, 2*`) + 2` is valid, but `a(`*1, 2*`) = 3+2` is not). Future versions may support this, however, if there is demand for such functionality. *n* and *k* must be complex numbers with no imaginary components. *n* and *k* must also each be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* and *k* must each be at least 1 after rounding.

- `a(`n_1, n_2, n_3, ...`)` if `a` is not a variable containing a matrix, it is considered to be user-defined function call. n_1, n_2, and n_3 are function arguments that can be expressions or variables of any type. There is no constraint on the number of arguments. Also note that passing QuIDD arguments and QuIDD return values only requires $O(1)$ memory since only a single pointer to the head of a QuIDD needs to be passed. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used). See Section B.2 for more details.

- `atan(`*n*`)` returns the arc tangent of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise arc tangent of *n*.

- `cb("...")` returns a computational basis state vector. The string literal argument consists of a sequence of any number and combination of '0' and '1' characters. The string is parsed from left to right. Each '0' causes a $|0\rangle$ to be tensored into the vector, and each '1' causes a $|1\rangle$ to be tensored into the vector. `cb` can easily be used to create density matrices by using it in conjunction with the complex-conjugate transpose operation (`'`), matrix multiplication, and scalar operations.

- `cnot("...")` returns a 2-qubit controlled-NOT (CNOT) gate matrix. This is a faster, specialized version of `cu_gate`. If a controlled gate matrix with different numbers of controls/targets and/or a different action (U operator) is desired, then use the more general `cu_gate` function. The argument of `cnot` is a string literal using the same gate specification syntax as `cu_gate`. However, the only valid parameters accepted by `cnot` are *'cx'* and *'xc'*, since these string specifications are the only possible strings that produce a valid 2-qubit CNOT gate matrix. For example, `cnot('cx')` produces a CNOT gate matrix with the control on the "top" wire and the action (X operator) on the "bottom" wire. For a discussion of how the concept of wires relates to creating controlled gate matrices, see `cu_gate`.

- `conj(`*n*`)` returns the complex-conjugate of the expression *n* passed as an argument. *n* can be a complex number or a matrix.

- `cos(`*n*`)` returns the cosine of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise cosine of *n*.

- `cps(`*n*`)` returns an *n*-qubit conditional phase shift (CPS) gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own CPS matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly. The conditional phase shift gate is particularly useful in Grover's quantum search algorithm [33].

- `cu_gate(`*a*, "`...`"`)` is a generalized controlled-*U* gate matrix creation function. It returns a controlled or uncontrolled gate matrix given an action matrix (*a*) and a string literal with the gate specification (the second argument contained in "s). The string literal consists of a sequence of characters. The idea is for the string literal to specify what the gate should do to each "wire" in a quantum circuit. When conceptualizing a quantum circuit graphically and reading top-down, the first character corresponds to the first qubit wire, the second character corresponds to the second qubit wire, etc. Each character can take one of four possible values. 'i' denotes the identity, which means that the gate does nothing to the wire at that location. 'x' denotes an action, which means that the matrix specified by the argument *a* is applied to the wire at that location. 'c' denotes a control, which means that the wire at that location is used as a control on any 'x' wire (a $|1\rangle$ state forces *a* to operate on any 'x' wire, whereas a $|0\rangle$ causes nothing to happen on any 'x' wire). 'n' is a

negated control, which is the opposite of 'c' (a $|0\rangle$ state forces *a* to operate on any 'x' wire, whereas a $|1\rangle$ causes nothing to happen on any 'x' wire). Any sequence of these characters may be used. Although there is no "actual" circuit, the string characters allow a user to conceptualize a circuit and construct a matrix which operates on the wires in that conceptualized circuit. *a* may be a matrix that operates on more than one qubit as long as one or more blocks of contiguous 'x' characters appear such that the size of each block is equal to the number of qubits operated on by *a*. For examples, see steaneX.qpro and steaneZ.qpro under the examples/nist/ subdirectory. Always use this function instead of defining your own gates explicitly, since it is asymptotically faster and uses asymptotically less memory. Since `cu_gate` must parse the input specification string, other functions such as `hadamard` and `cps` should be used instead of `cu_gate` for specific gates because they do not perform any parsing and are therefore a bit more efficient. An alternative function name for `cu_gate` is `lambda`. Also see the alternative, condensed version of `cu_gate` discussed next. The alternative version may be preferable for circuits with many qubits.

- `cu_gate(`*a*, "...", *n*`)` is an alternative syntax for `cu_gate` which takes a condensed string literal "...". This condensed string literal specifies only the actions and controls along with the qubit wires they are applied to. For example, a Toffoli gate in a 5-qubit circuit, with controls on the second and fourth wires and the action on the fifth wire, can be created with the call `cu_gate(sigma_x(1), "c2c4x5", 5)`. As implied by this example, *n* is the total number of qubits in the circuit that the

gate is applied to. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. More examples can be found in the examples/ directory and include hadder_bf1.qpro and rc_adder1.qpro, among others.

- `cutoff_val` If the cutoff value is set, any portion of all QuIDD element values that is less than the cutoff value will be rounded. For example, `cutoff_val = 1e-15` will cause all subsequently created QuIDD element values to be rounded at the 15th decimal place. By default, the cutoff value is not set and no rounding occurs. If the cutoff value is set by the user, it can be reset to the default (i.e. no rounding) by assigning 0 to `cutoff_val`.

- `dump_dot("...", "...", a)` outputs the *dot* form of the graphical QuIDD representation of the matrix/vector *a* to a file specified by the second argument. The first argument is the name that will appear at the top of the QuIDD image. dot is a simple scripting language supported in the Graphviz package[1] Once the dot file is generated, dot can be run from the command line to produce a PostScript image of the QuIDD representation as such:

  ```
  dot -Tps filename.dot -o filename.ps
  ```

  dot can generate other graphical file formats as well. Consult Graphviz for more details. A simple example is contained in the examples/dot subdirectory.

[1] Graphviz can be obtained at `http://www.graphviz.org/`.

- `echo("...")` prints the string literal passed as an argument to standard output. Putting one or more semicolons after `echo` does not suppress its output. `echo` has no return value, so it cannot be used in expressions.

- `else` is a program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. Only one `else` may optionally appear in an "if-elseif-else" block, and it must appear only at the end of the block. If an `else` block is used, its body (a sequence of zero or more expressions and/or control blocks to be executed) must be terminated by an `end` even if the body is empty. The body following `else` is executed when the preceding `if` and `elseif` conditions evaluate to "false" (i.e. a complex numbered value of zero).

- `elseif` is a program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. Zero or more `elseif`'s may appear in an "if-elseif-else" block, but the first `elseif` must appear after an `if`, and the last `elseif` must appear before an optional `else`. If no `else` appears after an `elseif`, the body of the `elseif` (a sequence of zero or more expressions and/or control blocks to be executed) must be terminated by an `end` even if the body is empty. The condition determines whether or not the statements in the body are executed. The body of the `elseif` is executed when the following two conditions are met: 1.) the preceding `if` and `elseif` conditions evaluate to "false" (i.e. a complex numbered value of zero), and 2.) the `elseif` condition evaluates to "true" (i.e. any non-zero complex numbered value).

- `end` keyword that signifies the end of a program flow control construct. In other words, `end` should be used to denote the end of "if-elseif-else" and "while" blocks.

- `exp(`n`)` returns e^n. If n is a matrix, then it returns a matrix containing the element-wise computation of e^k where k is an element from n.

- `expm(`n`)` returns e^n, where n is a matrix. This is standard matrix exponentiation and is approximated by a finitely bounded Taylor series. In the current version of the QuIDDPro simulator, you may only apply `expm` to a matrix n whose dimensions do not exceed 8×8 for efficiency reasons. Future versions may support larger dimensional arguments, but it is unlikely that larger dimensional arguments will be needed for most quantum-mechanics applications. If n is a complex number, then it returns e^n.

- `eye(`n`)` returns an $n \times n$ identity matrix. If you only need an identity matrix whose dimensions are a power of 2 in size (e.g. for k-qubit identity gate matrices) then use $identity(k)$ instead (see below), which runs slightly faster. n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding. Always use `eye` or `identity` instead of defining identity matrices explicitly because they are asymptotically faster and use asymptotically less memory.

- `fredkin()` returns a Fredkin gate matrix.

- `function` *var_name* = *func_name*(*n*$_1$, *n*$_2$, *n*$_3$, ...) defines a function body. This definition should exist in a file by itself with a filename that matches *func_name* appended by the ".qpro" or ".qp" extensions. *var_name* is the name of the variable that contains the return value. n_1, n_2, and n_3 are function parameters that can be of any type. There is no constraint on the number of parameters. Also note that passing QuIDD arguments and QuIDD return values only requires $O(1)$ memory since only a single pointer to the head of a QuIDD needs to be passed. Arguments passed as parameters to functions are not modified by the function (i.e. pass-by-value is always used). Following the return value/function name line, the script code comprising the function body should appear. See Section B.2 for more details.

- `gen_amp_damp`(*d, p, n, a*) performs generalized amplitude dampening (see [51, p. 382] for a description of generalized amplitude dampening). *a* is a density matrix (it must be square and have dimensions that are a power of 2 in size) on which dampening is to be performed. *a* is not modified, but the result of dampening applied to *a* is returned. *d* is the dampening parameter and must be a complex number with no imaginary component. *p* is the probability parameter and must also be a complex number with no imaginary component. *d* and *p* must each be in the range $[0,1]$. *n* is the qubit wire number that dampening is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of

an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `hadamard(n)` or `H(n)` returns an *n*-qubit Hadamard gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own Hadamard matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `i` is a variable that is preset to the value $0+1i$. It can be overwritten at runtime by the user.

- `identity(n)` returns an *n*-qubit identity gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own identity matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly. Also see the *eye* function.

- `if` is a program flow control construct that is part of an "if-elseif-else" control block sequence. Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. An "if-elseif-else" block must be started by a single `if`, but "if-elseif-else" blocks can be nested within other "if-elseif-else" blocks (nesting with "while" blocks is also allowed). An `if` must be followed by a body of zero or more expressions and/or control blocks, and this body must be terminated by either an `elseif`, an `else`, or an `end`, even if the body is empty. The condition determines whether or not the statements in the body are executed. The body is executed once if the condition evaluates to "true" (i.e. any non-zero complex numbered value). Otherwise if the condition evaluates to "false" (i.e. a complex numbered value of zero), the body is not executed.

- `kron(`*n*, *k*`)` returns the tensor (Kronecker) product of the matrix expressions *n* and *k*. If *n* and *k* are complex numbers, then they are multiplied together.

- `lambda(`*a*, "`...`"`)` is an alternative name for the function `cu_gate`.

- `measure(`*n*, *a*`)` performs deterministic measurement on the *n*th qubit in the density matrix *a*. In other words, all off-diagonal correlations corresponding to the qubit being measured are zeroed out, and the resultant density matrix is returned (for probabilistic measurement of a qubit in a density matrix that returns a 1 or 0, see `pmeasure`). *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of measurement applied to *a* is returned. *n* is the qubit wire number that measurement is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in

the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `measure_sv(`*n, a*`)` perform probabilistic measurement on qubit *n*. A state vector is returned which represents the state vector *a* as modified by the measurement result and its associated norm. If the measurement result and the associated norm have already been computed with a previous call to `pmeasure_norm_sv`, then `measure_sv` can be called with the alternative syntax `measure_sv(`*n, a, res, norm*`)`. *res* and *norm* denote the precomputed measurement result and associated norm, respectively. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. *res* must have the value 0 or 1 to within the rounding threshold. *norm* should be a valid norm of a state vector.

- `measure_sv0(`*n, a*`)` returns the probability of measuring qubit *n* as a 0 in state vector *a* (for probabilistic measurement of a qubit in a state vector that returns a

1 or 0, see `pmeasure_sv`). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `measure_sv1(`*n*, *a*`)` returns the probability of measuring qubit *n* as a 1 in state vector *a* (for probabilistic measurement of a qubit in a state vector that returns a 1 or 0, see `pmeasure_sv`). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `norm(`*a*`)` returns the norm of a state vector or complex number *a*. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2.

- `output_prec` denotes the output precision. When assigned a non-negative integer value, it specifies how many digits should be output to the screen. Any digits which exceed this number are rounded. For example, `output_prec = 3` will cause $1/3$ to output 0.333 to the screen. Note that the internal precision of any numbers and variables are unaffected. `output_prec` only affects the screen output precision. By

default, the variable `output_prec` is not set, but the output precision is initially 6. Assigning a negative value to `output_prec` restores the default output precision. However, assigning a matrix to `output_prec` leaves the precision unchanged from its previous value.

- `pi` is a variable that is preset to the value of π to a large number of decimal places. It can be overwritten at runtime by the user.

- `pmeasure(`*n*`,` *a*`)` performs probabilistic measurement on the *n*th qubit in the density matrix *a*. The result returned is a 1 or 0 (for deterministic measurement of a qubit in a density matrix, see `measure`). *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `pmeasure_norm_sv(`*n*`,` *a*`)` performs probabilistic measurement on the *n*th qubit in the state vector *a*. A 1×2 vector is returned containing a 1 or 0 for the measurement result (the first element) and the norm associated with the measurement result (the second element). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g.

9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `pmeasure_sv(`*n*, *a*`)` performs probabilistic measurement on the *n*th qubit in the state vector *a*. The result returned is a 1 or 0 (for deterministic measurement of a qubit in a state vector see `measure_sv0` and `measure_sv1`). Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `proj0(`*n*`)` returns an *n*-qubit $|0\rangle$ projector gate matrix (i.e. $|0\ldots0\rangle\langle0\ldots0|$, for *n* 0's). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|0\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `proj1(`*n*`)` returns an *n*-qubit $|1\rangle$ projector gate matrix (i.e. $|1\ldots1\rangle\langle1\ldots1|$, for *n* 1's). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g.

9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|1\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `projplus(n)` returns an *n*-qubit $|+\rangle$ projector gate matrix (i.e. $|+\ldots+\rangle\langle+\ldots+|$, for *n* +'s). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own $|+\rangle$ projector matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `ptrace(n, a)` performs the partial trace over the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of the partial trace applied to *a* is returned. *n* is the qubit wire number that is traced over. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must

be at least 1 after rounding.

- `px(p, n, a)` applies a probabilistic Pauli X gate matrix to the nth qubit in the density matrix a. a must be square and have dimensions that are a power of 2 in size. a is not modified, but the result of dampening applied to a is returned. p is the probability parameter and must be a complex number with no imaginary component. p must be in the range $[0, 1]$. n is the qubit wire number that the probabilistic X gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the nth quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). n must be a complex number with no imaginary component. n must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, n must be at least 1 after rounding.

- `Px(p, n, a)` an alternative name for the function `px`.

- `py(p, n, a)` applies a probabilistic Pauli Y gate matrix to the nth qubit in the density matrix a. a must be square and have dimensions that are a power of 2 in size. a is not modified, but the result of dampening applied to a is returned. p is the probability parameter and must be a complex number with no imaginary component. p must be in the range $[0, 1]$. n is the qubit wire number that the probabilistic Y gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the nth quantum state in the density matrix (see *cu_gate* for a more detailed de-

scription of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `Py(p, n, a)` an alternative name for the function `py`.

- `pz(p, n, a)` applies a probabilistic Pauli *Z* gate matrix to the *n*th qubit in the density matrix *a*. *a* must be square and have dimensions that are a power of 2 in size. *a* is not modified, but the result of dampening applied to *a* is returned. *p* is the probability parameter and must be a complex number with no imaginary component. *p* must be in the range $[0,1]$. *n* is the qubit wire number that the probabilistic *Z* gate matrix is to be applied to. This wire number is only conceptual and can alternatively be thought of as the *n*th quantum state in the density matrix (see *cu_gate* for a more detailed description of wire numbers and steaneX.qpro and steaneZ.qpro under examples/nist/ for examples). *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding.

- `Pz(p, n, a)` an alternative name for the function `pz`.

- `qp_epsilon` This checks an internal cache when creating new QuIDD element values, a cache is checked internally to see if those values have already been created. The more repeated values there are in a matrix, the more the matrix is compressed by its QuIDD representation. When checking the cache, QuIDDPro compares the equality of a new value to other values already in the cache to using an epsilon. Specifically, a and b are considered equal if $abs(a-b) < epsilon * a$ and $abs(a-b) < epsilon * b$. Epsilon can be changed by assigning values to `qp_epsilon`. By default, the epsilon value is $1e-8$. Currently, the epsilon value is not always used when creating new QuIDD element values, but in future versions of QuIDDPro, the epsilon value will play a much greater role.

- `quidd_info(a)` prints information about an operator or state to standard output. This information includes the number of qubits represented (or acted upon), the dimensions of the explicit representation of the matrix, and the number of nodes in the QuIDD representation of the matrix. Note that the explicit matrix representation is not actually stored anywhere. a must be a valid operator, state vector, or density matrix.

- `r2` is a variable that is preset to the value of $\sqrt{2}$ to a large number of decimal places. It can be overwritten at runtime by the user.

- `r3` is a variable that is preset to the value of $\sqrt{3}$ to a large number of decimal places. It can be overwritten at runtime by the user.

- `rand(n)` returns a pseudo-random value between 0 and *n*. *n* can be any real value, including negative values.

- `round(n)` returns *n* with its real and imaginary parts rounded to the nearest integer. "Halfway" cases are rounded away from 0. Since there is no native integer type supported in QuIDDPro, round can be extremely helpful in ensuring that values which are supposed to be integer values are indeed integer values.

- `run "..."` executes all script code contained in the file specified by the argument. The run command may appear anywhere in a script except inside an explicit matrix. This command is ideal for declaring variables that may be re-used in multiple projects.

- `rx(n, k)` returns a *k*-qubit rotational Pauli *X* gate matrix given a real valued angle parameter *n*. *n* must be a complex number with no imaginary component. *n* must be in the range $[0,1]$. *k* must be a complex number with no imaginary component. *k* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *k* must be at least 1 after rounding.

- `Rx(n, k)` is an alternative name for the function `rx`.

- `ry(n, k)` returns a *k*-qubit rotational Pauli *Y* gate matrix given a real valued angle parameter *n*. *n* must be a complex number with no imaginary component. *n* must be in the range $[0,1]$. *k* must be a complex number with no imaginary component.

k must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *k* must be at least 1 after rounding.

- `Ry(n, k)` is an alternative name for the function `ry`.

- `rz(n, k)` returns a *k*-qubit rotational Pauli *Z* gate matrix given a real valued angle parameter *n*. *n* must be a complex number with no imaginary component. *n* must be in the range $[0,1]$. *k* must be a complex number with no imaginary component. *k* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *k* must be at least 1 after rounding.

- `Rz(n, k)` is an alternative name for the function `rz`.

- `sigma_x(n)` returns an *n*-qubit Pauli *X* gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own *X* matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `sigma_y(n)` returns an *n*-qubit Pauli *Y* gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own *X* matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `sigma_z(n)` returns an *n*-qubit Pauli *Z* gate matrix. *n* must be a complex number with no imaginary component. *n* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* must be at least 1 after rounding. Always use this function instead of explicitly defining your own *X* matrix. This function is asymptotically faster and uses asymptotically less memory than defining the matrix explicitly.

- `sin(n)` returns sine of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise sine of *n*.

- `tan(n)` returns the tangent of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise sine of *n*.

- `sqrt(n)` returns the square root of the expression *n* passed as an argument. If *n* is a matrix, it returns a matrix containing the element-wise square root of *n*.

- `swap(`*n*, *k*, *a*`)` returns the vector resulting from swapping qubits *n* and *k* in the state vector *a*. This function swaps qubits *much more quickly* than swapping using CNOT and Hadamard gates. Since *a* must be a state vector, one of the dimensions must be 1, and the other dimension must be a power of 2. *a* is not modified by this function. *n* and *k* must be complex numbers with no imaginary components. *n* and *k* must also be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). *n* and *k* must also be at least 1 after rounding.

- `tic` starts a timer and also starts to record the peak memory usage from the point `tic` is called. `tic` has no return value, so it cannot be used in expressions. The timer only records time spent and memory used while running code. Thus, in the case of interactive mode, the timer will not be recording time spent nor memory used while at an idle prompt.

- `toc` stops a timer started by a previous `tic` or `toc` command. It outputs to standard output the time that has elapsed (i.e. time spent running code), the number of gates applied, the average runtime per gate, and memory that was used (peak memory) since the last `tic` or `toc` command. It also outputs the base memory which is the memory used in initializing the simulator and reading the input code. Base memory should be interpreted as a one-time initialization cost of the simulator and should not be considered when measuring performance. Operations that are recorded as applied gates include matrix multiplication, `gen_amp_damp`, `measure`, `measure_sv`,

`Px`, `Py`, and `Pz`.

- `toffoli("...")` returns a 3-qubit Toffoli gate matrix. This is a faster, specialized version of `cu_gate`. If a controlled gate matrix with different numbers of controls/targets and/or a different action (*U* operator) is desired, then use the more general `cu_gate` function. The string argument uses the same syntax as that of `cu_gate`. However, `toffoli` only accepts the strings *'ccx'*, *'cxc'*, and *'xcc'*, since these are the only valid Toffoli specifications. For example, `toffoli('ccx')` produces a Toffoli gate matrix with the controls on the "top" two wires and the action (*X* operator) on the "bottom" wire. For a discussion of how the concept of wires relates to creating controlled gate matrices, see `cu_gate`.

- `while` is a program flow control construct that allows multiple iterations of a body of code ("looping"). Its meaning is the same as in just about any other language. It contains a condition which is an expression enclosed in parentheses. A "while" block must be started by a single `while`, but "while" blocks can be nested within other "while" blocks (nesting with "if-elseif-else" blocks is also allowed). A `while` must be followed by a body of zero or more expressions and/or control blocks, and this body must be terminated by an `end`, even if the body is empty. The condition determines whether or not the statements in the body are executed. As long as the condition evaluates to "true" (i.e. any non-zero complex numbered value), the body is iteratively executed. The iterations stop when the condition becomes "false" (i.e. a complex numbered value of zero). The condition is checked once prior to executing each iteration of the body. `for` loops are also implemented with the counter variable,

termination condition, and incrementing expression separated by commas.

- `zeros(`*n*, *k*`)` returns an $n \times k$ matrix of all 0's. *x* and *y* must be complex numbers with no imaginary components. *n* and *k* must be complex numbers with no imaginary component. *n* and *k* must also each be within $10E-5$ of an integer value (e.g. 9.99999, 1.00001, and 3 are valid, but 4.5 is not), and values that are within this threshold are rounded to the nearest integer (e.g. 9.99999 is interpreted as 10). In addition, *n* and *k* must each be at least 1 after rounding. Always use *zeros* instead of defining zero matrices explicitly because it is asymptotically faster and uses asymptotically less memory.

APPENDIX C

QuIDDPro Examples

This appendix provides several sample QuIDDPro implementations of various quantum circuits. Section C.1 contains small examples which create three well-known quantum states. Sections C.2 and C.3 offer larger examples which implement Grover's quantum search algorithm [33] and Shor's quantum integer factoring algorithm [65], respectively.

C.1 Well-known Quantum States

This section contains QuIDDPro code which implements the cat (GHZ) state, the W state, and the equal superposition state. These examples illustrate how the language can be used to produce code that is as compact as the formal definition of such states.

C.1.1 Cat State

The cat state is an n-qubit generalization of the EPR pair and is defined as $|\psi_{\text{cat}}\rangle = (|00\ldots0\rangle + |11\ldots1\rangle)/\sqrt{2}$. A QuIDDPro function which creates this state given the number of qubits n is listed below.

```
function |cs:> = create_cat_state(n)

   |cs:> = (|0:>_n + |2^n - 1:>)/sqrt(2);
```

C.1.2 W State

The W state is an *n*-qubit state defined as $|\psi_W\rangle = (|10\ldots0\rangle + |01\ldots0\rangle + |00\ldots1\rangle)/\sqrt{n}$. A QuIDDPro function which creates this state given the number of qubits *n* is given below.

```
function |ws:> = create_w_state(n)

   |ws:> = |1:>_n;

   for (j = 1, j < n, j++)

      |ws:> += |2^j:>;

   end

   |ws:> /= sqrt(n);
```

C.1.3 Equal Superposition State

The equal superposition state is an *n*-qubit state which represents all possible 2^n measurement outcomes with equal probability. It is defined as $\frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle$ and can be created with Hadamard gates. A QuIDDPro function which creates this state given the number of qubits *n* is provided below.

```
function |sps:> = create_equal_superposition(n)

   |sps:> = H(n)*|0:>_n;
```

C.2 Grover's Search Algorithm

This section demonstrates how Grover's quantum search algorithm can be implemented in QuIDDPro. The first function provided below takes as arguments the number of qubits *n*, the oracle defining the search criteria, and an estimated number of items in the database

which match the search criteria. This function returns the integer representation of the index measured of an item in the database (the left-most qubit in the state is most-significant). One ancillary qubit is used in conjunction with the oracle. As noted in the last appendix, assignment of QuIDDs only requires $O(1)$ time, which means that passing the oracle QuIDD to this function involves very little overhead. Another function is included later in this section which creates an oracle that searches for the last item in the database (the item with index $|11\ldots1\rangle$) which can be used in conjunction with the function implementing Grover's algorithm.

```
function index = grover_search(n, oracle, matches)
   |state:> = H(n)*|1:>_(n + 1);
   grover_op = H(n)*cps(n)*H(n)*oracle;
   # Compute the optimal number of Grover iterations.
   N = 2^n;
   x = sqrt(matches/N);
   theta = atan(x/sqrt(1 - x^2));
   num_iterations = pi/4/theta;
   # Perform the Grover iterations.
   for (g = 0, g < num_iterations, g++)
      |state:> = grover_op*|state:>;
   end
   # Measure an index.
   index = 0;
```

```
   for (q = 1, q <= n, q++)
      if (pmeasure_sv(q, |state:>))
         index += 2^(n - q);
      end
   end


function oracle = create_last_item_oracle(n)
   oracle_spec = ‘‘x’’;
   for (j = 0, j < n, j++)
      oracle_spec = ‘‘c’’ + oracle_spec;
   end
   oracle = cu_gate(sigma_x(1), oracle_spec);
```

C.3 Shor's Integer Factoring Algorithm

This section demonstrates a possible implementation of the main portion of Shor's algorithm. Given an integer N and its size in bits n, the following function uses quantum order-finding to find a non-trivial factor of N. Order-finding solves the problem of determining r such that $a^r \ mod \ N = 1$. For the purposes of factoring, a may be chosen randomly from the range $(1..N)$, and in the following function it is simply passed as an argument. Quantum modular exponentiation is used to compute all possible values for x and $a^x \ mod \ N$ simultaneously. Following this step, the inverse QFT is applied to increase the probability of measuring qubit values for which the state representation of $a^x \ mod \ N$

encodes the value 1 in binary. The value for x that is entangled with this part of the state is r. Classical post-processing is shown at the end, which makes use of the greatest common divisor algorithm. Not shown are functions implementing quantum modular exponentiation and the inverse QFT, each of which can be implemented in a variety of different ways [74, 73, 28].

```
function factor = shor_factor(N, a, n)
   if (N{1} == 0)
      factor = 2;
   else
      # Put the exponent state into an equal superposition.
      |x:> = H(n)*|0:>_n;
      |mod:> = |1:>_n;
      # Compute modular exponentiation and the inverse QFT.
      |res:> = mod_exp(|x:>, |mod:>, N, a, n);
      |res:> = inv_qft(|res:>, n);
      # Measure the exponent qubits.
      r = 0;
      for (q = 1, q <= n; q++)
         if (pmeasure_sv(q, |res:>))
            r += 2^(n - q);
         end
      end
```

```
   # Check if r can be used to calculate a factor.
   if ((r{1} == 0) && (rem(a^(r/2), N) != 1))
      cand_fac1 = gcd(a^(r/2) - 1, N);
      cand_fac2 = gcd(a^(r/2) + 1, N);
      if (rem(N, cand_fac1) == 0)
         factor = cand_fac1;
      elseif (rem(N, cand_fac2) == 0)
         factor = cand_fac2;
      else
         factor = -1;
      end
   else
      factor = -1;
   end
end
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Aaronson and D. Gottesman, "Improved Simulation of Stabilizer Circuits," *Phys. Rev. A* **70**, 052328, 2004.

[2] D. Aharonov, Z. Landau, and J. Makowsky, "The Quantum FFT can be Classically Simulated," `quant-ph/0611156`, 2006.

[3] S. Anders and H. J. Briegel, "Fast Simulation of Stabilizer Circuits Using a Graph State Representation," *Phys. Rev. A* **73**, 022334, 2006.

[4] R. I. Bahar et al., "Algebraic Decision Diagrams and their Applications," *Journal of Formal Methods in System Design* **10** (2/3), pp. 171-206, 1997.

[5] A. Barenco et al., "Elementary Gates for Quantum Computation," *Phys. Rev. A* **52**, pp. 3457-3467, 1995.

[6] C. H. Bennett and G. Brassard, "Quantum Cryptography: Public Key Distribution and Coin Tossing," *In Proc. of IEEE Intl. Conf. on Computers, Systems, and Signal Processing*, pp. 175-179, 1984.

[7] C.H. Bennett, "Quantum Cryptography Using Any Two Nonorthogonal States," *Phys. Rev. Lett.* **68**, pp. 3121-3124, 1992.

[8] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an Unknown Quantum State via Dual Classical and Einstein-Podolsky-Rosen Channels," *Phys. Rev. Lett.* **70**, 1895, 1993.

[9] G. P. Berman, G. D. Doolen, G. V. López, and V. I. Tsifrinovich, "Simulations of Quantum-logic Operations in a Quantum Computer with a Large Number of Qubits," *Phys. Rev. A* **61**, 062305, 2000.

[10] G.P. Berman et al., "Analytic Solutions for Quantum Logic Gates and Modeling Pulse Errors in a Quantum Computer with a Heisenberg Interaction," *International Journal of Quantum Information* **2** (2), pp. 171-182, 2003.

[11] G. P. Berman, G. V. López, and V. I. Tsifrinovich, "Teleportation in a Nuclear Spin Quantum Computer," *Phys. Rev. A* **66**, 042312, 2002.

[12] P. E. Black et al., *Quantum Compiling and Simulation*, `http://hissa.nist.gov/~black/Quantum/`.

[13] B. M. Boghosian and W. Taylor, "Simulating Quantum Mechanics on a Quantum Computer," *Physica D* **120**, pp. 30-42, 1998.

[14] M. Boyer, G. Brassard, P. Hoeyer and A. Tapp, "Tight Bounds on Quantum Searching," *Fortsch. Phys.* **46**, pp. 493-506, 1998.

[15] G. K. Brennen, "Distant Entanglement with Nearest Neighbor Interactions," `quant-ph/0206199`, 2002.

[16] H. J. Briegel and R. Raussendorf, "Persistent Entanglement in Arrays of Interacting Particles," *Phys. Rev. Lett.* **86**, pp. 910-913, 2001.

[17] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers* **C35**, pp. 677-691, 1986.

[18] B. Butscher and H. Weimer, "libquantum: the C Library for Quantum Computing," `http://www.enyo.de/libquantum/`.

[19] G. L. Celardo, C. Pineda, M. Znidaric, "Stability of Quantum Fourier Transformation on Ising Quantum Computer," `quant-ph/0310163`, 2003.

[20] J. Chiaverini et. al, "Realization of Quantum Error Correction," *Nature* **432**, pp. 602-605, 2004.

[21] A. M. Childs, H. L. Haselgrove, and M. A. Nielsen, "Lower Bounds on the Complexity of Simulating Quantum Gates," *Phys. Rev. A* **68**, 052311, 2003.

[22] E. Clarke et al., "Multi-Terminal Binary Decision Diagrams and Hybrid Decision Diagrams," in T. Sasao and M. Fujita, eds, *Representations of Discrete Functions*, pp. 93-108, Kluwer, 1996.

[23] A. Ekert and P. L. Knight, "Entangled Quantum Systems and the Schmidt Decomposition," *Am. J. Phys.* **63** (5), pp. 415-423, 1995.

[24] E. Clarke, M. Fujita, P. C. McGeer, K. McMillan, and J. Yang, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *IWLS '93*, pp. 6a:1-15, May 1993.

[25] CNET News, *Start-up Makes Quantum Leap in Cryptography*, CNET News.com, November 6, 2003.

[26] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A New Quantum Ripple-carry Addition Circuit," `quant-ph/0410184`, 2004.

[27] A.K. Ekert, "Quantum Cryptography Based on Bell's Theorem," *Phys. Rev. Lett.* **67**, pp. 661-663, 1991.

[28] A. G. Fowler, S. J. Devitt, and L. C. L. Hollenberg, "Implementation of Shor's Algorithm on a Linear Nearest Neighbour Qubit Array," *Quantum Information and Computation* **4**, pp. 237-251, 2004.

[29] A. G. Fowler, C. D. Hill, and L. C. L. Hollenberg, "Quantum-error Correction on Linear-nearest-neighbor Qubit Arrays," *Phys. Rev. A* **69**, 042314, 2004.

[30] "GNU MP (GMP): Arithmetic Without Limitations," `http://www.swox.com/gmp/`

[31] D. Gottesman, "The Heisenberg Representation of Quantum Computers," *Plenary speech at the 1998 International Conference on Group Theoretic Methods in Physics*, `http://www.arxiv.org/abs/quant-ph/9807006`, 1998.

[32] D. Greve, "QDD: A Quantum Computer Emulation Library," `http://thegreves.com/david/QDD/qdd.html`, 1999

[33] L. Grover, "Quantum Mechanics Helps In Searching For A Needle In A Haystack," *Phys. Rev. Lett.* **79**, pp. 325-328, 1997.

[34] J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, 1993.

[35] A. J. G. Hey, ed., *Feynman and Computation: Exploring the Limits of Computers*, Perseus Books, 1999.

[36] G. Jaroszkiewicz, "Quantum Register Physics," `quant-ph/0409094`, 2004.

[37] R. Jozsa and N. Linden, "On the Role of Entanglement in Quantum Computational Speed-up," `quant-ph/0201143`, 2002.

[38] R. Jozsa, "On the Simulation of Quantum Circuits," `quant-ph/0603163`, 2006.

[39] K. Khodjasteh and D. A. Lidar, "Fault-tolerant Quantum Dynamical Decoupling," *Phys. Rev. Lett.* **95**, 180501, 2005.

[40] D. Kielpinski, V. Meyer, M. A. Rowe, C. A. Sackett, W. M. Itano, C. Monroe, and D. J. Wineland, "A Decoherence-free Quantum Memory Using Trapped Ions," *Science* **291**, pp 1013-1015, 2001.

[41] D. Kielpinski, C. Monroe, and D. J. Wineland, "Architecture for a Large-scale Ion-trap Quantum Computer," *Nature* **417**, pp. 709-711, 2002.

[42] A. Y. Kitaev, "Quantum Computations: Algorithms and Error Correction," *Russ. Math. Surv.* **52** (6), pp. 1191-1249, 1997.

[43] A. Y. Kitaev, A. H. Shen, and M. N. Vyalyi, *Classical and Quantum Computation*, American Mathematical Society, Graduate Studies in Mathematics, **47**, 2002.

[44] T. D. Ladd, J. R. Goldman, F. Yamaguchi, and Y. Yamamoto, "All-silicon Quantum Computer," *Phys. Rev. A* **89**, 017901, 2002.

[45] C. Y. Lee, "Representation of Switching Circuits by Binary Decision Diagrams," *Bell System Tech. J.* **38**, pp. 985-999, 1959.

[46] D. A. Lidar and L. A. Wu, "Quantum Computers and Decoherence: Exorcising the Demon from the Machine," quant-ph/0302198, 2003.

[47] S. Lloyd, "Universal Quantum Simulators," *Science* **273** (5278), pp. 1073-1078, 1996.

[48] I. L. Markov and Y. Shi, "Simulating Quantum Computation by Contracting Tensor Networks," quant-ph/0511069, 2005.

[49] D. Maslov, G. Dueck, and N. Scott, "Reversible Logic Synthesis Benchmarks Page," http://www.cs.uvic.ca/~dmaslov/.

[50] C. Monroe, "Quantum Information Processing with Atoms and Photons," *Nature* **416**, pp. 238-246, 2002.

[51] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge Univ. Press, 2000.

[52] K. M. Obenland and A. M. Despain, "A Parallel Quantum Computer Simulator," *High Performance Computing*, 1998.

[53] B. Ömer, "QCL - A Programming Language for Quantum Computers," http://tph.tuwien.ac.at/~oemer/qcl.html.

[54] *Open Qubit Quantum Computing*, http://www.ennui.net/~quantum/.

[55] D. Petrosyan and G. Kurizki, "Scalable Solid-state Quantum Processor Using Subradiant Two-atom States," *Phys. Rev. Lett.* **89**, 207902, 2002.

[56] A. K. Prasad, V. V. Shende, K. N. Patel, I. L. Markov, and J. P. Hayes, "Algorithms and data structures for simplifying reversible circuits," to appear in *ACM J. of Emerging Technologies in Computing*, 2007.

[57] V. Protopopescu, R. Perez, C. D'Helon, and J. Schmulen, "Robust Control of Decoherence in Realistic One-qubit Quantum Gates," *J. Phys. A: Math. Gen.* **36**, pp. 2175-2189, 2003.

[58] *QuIDDPro: High-Performance Quantum Circuit Simulation*, http://vlsicad.eecs.umich.edu/Quantum/qp/.

[59] R. Shankar, *Principles of Quantum Mechanics 2nd Ed.*, Plenum Press, 1994.

[60] V. V. Shende, S. S. Bullock, and I. L. Markov, "A Practical Top-down Approach to Quantum Circuit Synthesis," *In Proc of the Asia South Pacific Design Automation Conf.* (ASPDAC), pp. 272-275, 2005.

[61] V. V. Shende, S. S. Bullock, I. L. Markov, "Synthesis of Quantum Logic Circuits," *IEEE Trans. on Computer-Aided Design* **25**, pp. 1000-1010, 2006.

[62] V. V. Shende and I. L. Markov, "Quantum Circuits for Incompletely Specified Two-qubit Operators," *Quantum Information and Computation* **5** (1), pp. 49-57, 2005.

[63] V. V. Shende, A. K. Prasad, I. L. Markov and J. P. Hayes, "Synthesis of Reversible Logic Circuits," *IEEE Trans. on Computer-Aided Design*, **22** (6), pp. 710-722, 2003.

[64] Y. Shi, L. Duan, and G. Vidal, "Classical Simulation of Quantum Many-body Systems with a Tree Tensor Network," *Phys. Rev. A* **74**, 022320, 2006.

[65] P. W. Shor, "Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. of Computing* **26**, pp. 1484-1509, 1997.

[66] F. Somenzi, "CUDD: CU Decision Diagram Package," ver. 2.4.0, Univ. of Colorado at Boulder, 1998.

[67] G. Song and A. Klappenecker, "Optimal Realizations of Simplified Toffoli Gates," **4**, pp. 361-372, 2004.

[68] R. T. Stanion, D. Bhattacharya, and C. Sechen, "An Efficient Method for Generating Exhaustive Test Sets," *IEEE Trans. on Computer-Aided Design* **14**, pp. 1516-1525, 1995.

[69] A. M. Steane, "Error-correcting Codes in Quantum Theory," *Phys. Rev. Lett.*, **77**, p. 793, 1996.

[70] G. Strang, *Linear Algebra and its Applications*, Harcourt College Publishers, 1988.

[71] L. Tian and P. Zoller, "Quantum Computing with Atomic Josephson Junction Arrays," `quant-ph/0306085`, 2003.

[72] L. G. Valiant, "Quantum Computers that can be Simulated Classically in Polynomial Time," *Proc. of ACM Symp. on Theory of Computing* (STOC), pp. 114-123, 2001.

[73] R. Van Meter and K. M. Itoh, "Fast Quantum Modular Exponentiation," *Phys. Rev. A* **71**, 052320, 2005.

[74] V. Vedral, A. Barenco, and A. Ekert, "Quantum Networks for Elementary Arithmetic Operations," *Phys. Rev. A* **54**, pp. 147-153, 1996.

[75] T. Veldhuizen, "Arrays in Blitz++," *In Proc 2nd Intl. Symp. on Computing in OO Parallel Environments*, `http://www.oonumerics.org/blitz/`, 1998.

[76] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Graph-based simulation of quantum computation in the density matrix representation," *Quantum Information and Computation* **5** (2), pp. 113-130, 2005.

[77] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Is Quantum Search Practical?" *Computing in Science and Engineering* **7** (4), pp. 22-30, 2005.

[78] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Graph-based Simulation of Quantum Computation in the Density Matrix Representation," *In Proc. of SPIE* **5436**, pp. 285-296, 2004.

[79] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "High-performance QuIDD-based Simulation of Quantum Circuits," *In Proc. of the Design, Automation and Test in Europe Conference (DATE)* **2**, pp. 1354-1355, 2004.

[80] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Improving Gate-level Simulation of Quantum Circuits," *Quantum Inf. Processing* **2** (5), pp. 347-380, 2003.

[81] G. F. Viamontes, I. L. Markov, and J. P. Hayes, "Checking Equivalence of Quantum States, Operators and Circuits," submitted for publication, *Quantum Information and Computation*, 2006.

[82] G. F. Viamontes, M. Rajagopolan, I. L. Markov, and J. P. Hayes, "Gate-level Simulation of Quantum Circuits," *In Proc. of ACM/IEEE Asia and South-Pacific Design Automation Conf.* (ASPDAC), pp. 295-301, 2003.

[83] G. F. Viamontes, M. Rajagopalan, I. L. Markov, and J. P. Hayes, "Gate-Level Simulation of Quantum Circuits," *In Proc. of the 6th Intl. Conference on Quantum Communication, Measurement, and Computing*, pp. 311-314, 2002.

[84] G. Vidal, "Efficient Classical Simulation of Slightly Entangled Quantum Computations," *Phys. Rev. Lett.* **91**, 147902, 2003.

[85] G. Vidal, "Efficient Simulation of One-dimensional Quantum Many-Body Systems," *Phys. Rev. Lett.* **93**, 040502, 2004.

[86] L. Viola, E. Knill, and S. Lloyd, "Dynamical Decoupling of Open Quantum Systems," *Phys. Rev. Lett.* **82**, pp. 2417-2421, 1999.

[87] L. Viola and S. Lloyd, "Dynamical Suppression of Decoherence in Two-state Quantum Systems," *Phys. Rev. A* **58**, pp. 2733-2744, 1998.

[88] L. Viola, S. Lloyd, and E. Knill, "Universal Control of Decoupled Quantum Systems," *Phys. Rev. Lett.* **83**, 4888, 1999.

[89] R. Vrijen et. al, "Electron-spin-resonance Transistors for Quantum Computing in Silicon-germanium Heterostructures," *Phys. Rev. A* **62**, 012306, 2000.

ABSTRACT

Efficient Quantum Circuit Simulation

by

George F. Viamontes

Co-Chairs: John P. Hayes and Igor L. Markov

Quantum-mechanical phenomena are playing an increasing role in information processing as transistor sizes approach the nanometer level, while the securest forms of communication rely on quantum data encoding. When they involve a finite number of basis states, these phenomena can be modeled as quantum circuits, the quantum analogue of conventional or "classical" logic circuits. Simulation of quantum circuits can therefore be used as a tool to evaluate issues in the design of quantum information processors. Unfortunately, simulating such phenomena efficiently is exceedingly difficult. The matrices representing quantum operators (gates) and vectors modeling quantum states grow exponentially with the number of quantum bits.

The information represented by quantum states and operators often exhibits structure that can be exploited when simulating certain classes of quantum circuits. We study the development of simulation methods that run on classical computers and take advantage

of such repetitions and redundancies. In particular, we define a new data structure for simulating quantum circuits called the quantum information decision diagram (QuIDD). A QuIDD is a compressed graph representation of a vector or matrix and permits computations to be performed directly on the compressed data. We develop a comprehensive set of algorithms for operating on QuIDDs in both the state-vector and density-matrix formats, and evaluate their complexity. These algorithms have been implemented in a general-purpose simulator program for quantum-mechanical applications called QuIDDPro. Through extensive experiments conducted on representative quantum simulation applications, including Grover's search algorithm, error characterization, and reversible circuits, we demonstrate that QuIDDPro is faster than other existing quantum-mechanical simulators such as the National Institute of Standards and Technology's QCSim program, and is far more memory-efficient. Using QuIDDPro, we explore the advantages of quantum computation over classical computation, simulate quantum errors and error correction, and study the impact of numerical precision on the fidelity of simulations. We also develop several novel algorithms for testing quantum circuit equivalence and compare them empirically. The QuIDDPro software is equipped with a user-friendly interface and is distributed with numerous example scripts. It has been used as a laboratory supplement for quantum computing courses at several universities.