QUANTUM VALUE GATE SIMULATOR

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Xin Chen

April 2003

ABSTRACT

QUANTUM VALUE GATE SIMULATOR

by Xin Chen

In this project, a polynomial implementation idea of simulating Špalek's algorithm is explored and implemented. The simulator supports ways of experiments with error models applied to the base gates. Experimental results are presented and evaluated. Also, the foundational concepts and notations of quantum computation used to understanding this work are introduced. The main results from Špalek's algorithm of simulating a value gate with small depth quantum circuits are reviewed.

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to my advisor, Dr. Chris Pollett, for introducing me to this exciting research area. This project would not have been possible without his guidance, patience and constant encouragement.

I would like to thank Dr. Robert Chun, and Dr. Walter Kirchherr for having made available their time and commitment to serve on my committee.

I would like to thank my parents for always supporting and encouraging me and for their unflinching faith in my abilities. Without their unselfish love and affection I would not have written this thesis.

I would like to thank Min Ouyang for his patience and support through the good dreams and the nightmares.

Contents

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 BACKGROUND **3**

CHAPTER 3 QUANTUM COMPUTATION **4**

3.1 MODELS OF COMPUTATION 4
3.2 FOUR POSTULATES OF QUANTUM MECHANICS 4
3.2.1 State space 4
3.2.2 Evolution 5
3.2.3 Quantum measurement 5
3.2.4 Composite systems 6
3.3 QUANTUM CIRCUIT 6
3.3.1 Single qubit operations 6
3.3.2 Quantum circuits 6

CHAPTER 4 ŠPALEK'S ALGORITHM **9**

4.1 QUANTUM FAN-OUT OPERATION 9
4.2 PARALLELIZATION METHOD 10
4.3 QUANTUM HADAMARD TRANSFORM 10
4.4 INCREMENT OPERATION 11
4.5 VALUE GATE 11

CHAPTER 5 DESIGN AND IMPLEMENTATION **13**

5.1 MATRIX REPRESENTATION 13
5.1.1 Matrix for the Hadamard layer 14
5.1.2 Matrix for the Permutation layer 15
5.1.3 Matrix for the Fan-out layer 16
5.1.4 Matrix for the Increment layer 17
5.1.5 Matrix for the entire circuit 18
5.2 A NAÏVE IMPLEMENTATION 19
5.2.1 Class organization 19
5.2.2 Limitation of the naïve implementation 21
5.3 A VERY EFFICIENT IMPLEMENTATION 21
5.3.1 Sufficient for reducing computational complexity 21
5.3.1.1 Determining which element is needed 22
5.3.1.2 Reducing the size of problem from exponential to polynomial 22
5.3.1.3 Getting a row of a layer matrix 27
5.3.2 Implementation detail 29
5.3.2.1 Saving each row of the layer matrix 29
5.3.2.2 Computing and saving the layer's matrix 30
5.3.2.3 Computing a intermediate row 32
5.3.2.4 Class organization 35
5.4 ERROR SCHEME 36

CHAPTER 6 TESTS 38

6.1 TEST CASE DESIGN 38
6.2 TEST RESULTS SAMPLE 40
6.3 TEST RESULTS ANALYSIS 50

CHAPTER 7 CONCLUSION 52

REFERENCES ……………………………………………………..53

List of Figures

Figure 3- 1 Circuit for the controlled-*U* operation 7
Figure 3- 2 Circuit and matrix representation for controlled-NOT operation 7
Figure 3- 3 Circuit for the $C^n(U)$ operation when n=3 and k=3 8
Figure 4- 1 The Hadamard transform $H^{\otimes 2}$ on two qubits 11
Figure 4- 2 Quantum circuit for value gate 12
Figure 5- 1 Layers of the circuit for value gate 14
Figure 5- 2 Permutation operator 15
Figure 5- 3 Fan-out layer transformation 16
Figure 5- 4 Matrix for *f* operator 17
Figure 5- 5 Increment layer transformation 17
Figure 5- 6 Matrix for $C^1(d)$ operator 18
Figure 5- 7 Class diagram 20
Figure 5- 8 Class diagram 36
Figure 6- 1 Test result for test 2.1 42
Figure 6- 2 Test result for test 2.2 42
Figure 6- 3 Test result for test 2.3 43
Figure 6- 4 Test result for test 2.4 43
Figure 6- 5 Test result for test 2.5 44
Figure 6- 6 Test result for test 2.6 44
Figure 6- 7 Test result for test 2.7 45
Figure 6- 8 Test result for test 2.8 45
Figure 6- 9 Test result for test 2.9 46
Figure 6- 10 Test result for test 3.1 46
Figure 6- 11 Test result for test 3.2 47
Figure 6- 12 Test result for test 3.3 47
Figure 6- 13 Test result for test 3.4 48
Figure 6- 14 Test result for test 3.5 48
Figure 6-15 Test results on HP workstation 49

List of Tables

Table 6- 1 Test results for test 1..41

Chapter 1 Introduction

In recent years interest in quantum computation has been steadily increasing. One reason for this is due to Shor's [S97] discovery of a polynomial time quantum algorithm for factoring, which is one of the strongest arguments in favor of the superiority of quantum computing models over classical ones. Since this discovery, many efforts have been made to find new, efficient quantum algorithms for classical problems and to develop quantum complexity theory. The goal of this research will be to develop a simulator, which will aid in understanding the robustness of certain quantum algorithms.

Špalek gives a way of simulating value gates with small depth quantum circuits in the exact acceptance model [HS03]. The simulation is an improvement over what can be done with classical AND, OR, NOT circuits. Yet, Špalek's algorithm assumes that we can perform certain rotation operation to arbitrary accuracy. So the question is how much error is introduced if we choose a more realistic acceptance criterion? Good formal estimates of this are somewhat difficult to directly calculate from the algorithm itself, so it would be interesting to do some simulations.

In this project we developed a program that simulates Špalek's algorithm, that is, simulates a quantum circuit that performs the function of a classical value gate. Then we added an error facility to the simulator, so that it supports ways of experimenting with error models applied to the base gates. The simulator is implemented in a very efficient way; in theory, it can work on any number of bits using reasonable time and space.

This report is organized as follows. Chapter 2 describes the background of value gate. In Chapter 3 we introduce the concepts and notations used in quantum computation

that relative to our simulation. We briefly review the main results of Špalek's algorithm in Chapter 4.

A naïve implementation of Špalek's algorithm takes exponential space and time, and thus is infeasible. We present our implementation idea, which reduce both the space and the time complexity from exponential to polynomial, in detail in Chapter 5. Then, in Chapter 6, we explain our test case, show the test results, and give some analysis. Finally, Chapter 7 concludes the report.

I would like to mention that this project work is based on an early version of [HS03]. That early version of paper assumes infinite precision of the gates. During the time of our project was done, Špalek added more ideas to his final published version. The main idea he added is to allow the use of a fixed set of one-qubit gates to construct rotation operations. However, this does not have much effect on our simulating.

Chapter 2 Background

A value gate is a logical gate that does the following computation: given an input with n bits and a threshold value m, if the number of bits that are '1' is equal to m, then the value gate outputs '1', otherwise the output is '0'. A value gate can be constructed by a classical AND-OR circuit [Vol00].

A threshold gate is a logical gate that performs the following logical function:

$$T_m(a_1, a_2, \ldots, a_n) =_{def} [\![\sum_{i=1}^{n} a_i > m]\!],$$

it can be simulated by the value gate circuits, that is, by circuits where the value gates are the only logical gates in the circuit.

Threshold gates play an important role in logical circuits. One example for this is shown in [Vol00], where an idea for constructing a constant-depth threshold circuits for multiplication is illustrated.

In this project, instead of simulating a value gate in a classical way, we simulate the value gate with a quantum circuit using Špalek's algorithm [HS03]. This quantum unit can be further embedded into the classical circuit to construct a threshold gate.

Chapter 3 Quantum Computation

Quantum mechanics is a mathematical model to describe the physics of the real world. In this section, we will review some important concepts of quantum mechanics, for details, see the textbook by Nielson and Chuang of quantum computation [NC00].

3.1 Models of Computation

In the classical world, we can use either a Turing machine or a logical circuit to represent the concept of a universal computer. Similarly, in quantum world, we can use either a quantum Turing machine or a quantum circuit to represent that same concept.

3.2 Four Postulates of Quantum Mechanics

There are four postulates of quantum mechanics, which are described below:

3.2.1 State space

For each isolated physical system, we can use a unit vector in a complex vector space with inner product to describe the system. This space is known as the state space of the system, the vector is known as the state vector. Quantum mechanics takes place in this state space.

A qubit is a two-dimensional state space. We can use $|0\rangle$ and $|1\rangle$ to form an orthonormal basis for the state space. Then any state in the state space can be written as:

$$|\psi\rangle = a|0\rangle + b|1\rangle,$$

where a and b are complex numbers, and $|a|^2 + |b|^2 = 1$.

3.2.2 Evolution

The evolution of a closed quantum system is described by a unitary transformation. That is, suppose at time t_1, the state of a system is $|\psi\rangle$, and at time t_2 it changes to state $|\psi'\rangle$. We can use a unitary operator U to describe this change, note that U depends only on the times t_1 and t_2,

$$|\psi'\rangle = U|\psi\rangle.$$

A transformation U is said to be unitary if $U^{\dagger}U = I$, where $U^{\dagger}$ is the conjugate-transpose of U. An example of unitary operator is the Hadamard operator, which we denote as H:

$$H = 1/\sqrt{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

3.2.3 Quantum measurement

We can use a collection of measurement operators, denoted as $\{M_m\}$, to describe the quantum measurements. Here m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement, then the probability that the measurement results in m is:

$$p(m) = \langle \psi|M_m^{\dagger} M_m|\psi\rangle,$$

and the system state right after the measurement is:

$$\frac{M_m|\psi\rangle}{\sqrt{\langle \psi|M_m^{\dagger} M_m|\psi\rangle}},$$

the measurement operators must satisfy the completeness equation:

$$\sum_m M_m^{\dagger} M_m = I.$$

3.2.4 Composite systems

If a quantum system is composed of several component systems, then its state space can be represented by the tensor product of the state space of these sub systems. For example, if a system is composed of *n* component systems, and the state of the *i*'th system is $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$.

3.3 Quantum Circuit

3.3.1 Single qubit operations

A single qubit is a vector $|\psi\rangle = a|0\rangle + b|1\rangle$, where *a* and *b* are complex numbers and must satisfy $|a|^2 + |b|^2 = 1$. Each operation (described by a 2x2 matrix) must be unitary to ensure the property that the probabilities must sum to one. We list two of the most important ones that are used in this project: the Hadamard gate (denoted *H*), and the Identity gate (denoted *I*) :

$$H = 1/\sqrt{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

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

3.3.2 Quantum circuits

- Controlled *U* operation

A controlled-*U* operation is a two qubits operation, as shown in Figure 3.1. In a controlled-*U* operation, there is a control qubit (the top line) and a target qubit (the

bottom line). If the control qubit is set, then the single qubit operation *U* is applied to the target qubit. If the control qubit is not set, then the target qubit will not change.

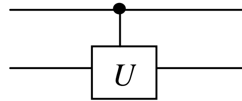
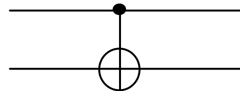


Figure 3- 1 Circuit for the controlled-*U* operation

- Controlled-NOT operation

A controlled-NOT operation is a special case of controlled-*U* operation. It flips the target qubit if the control qubit is set. Its circuit and matrix representation are shown in Figure 3-2.



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

Figure 3- 2 Circuit and matrix representation for controlled-NOT operation

- $C^n(U)$ operation

A controlled operation $C^n(U)$ is an operation on *n* control qubits and *k* target qubits, where *U* is a k qubit unitary operator. This operation can be defined by the following equation:

$$C^n(U)|x_1\, x_2 \dots x_n\rangle|\psi\rangle = |x_1\, x_2 \dots x_n\rangle\, U^{x1\, x2\, \dots\, xn}|\psi\rangle,$$

where $x_1\, x_2 \dots x_n$ is the product of the bits $x_1\, x_2 \dots x_n$. That is, when the *n* numbers of the control qubits are all set, the operator *U* is applied on the *k* target qubits, otherwise, the

target qubits are unchanged. The circuit notation for this operation is illustrated in Figure 3-3.

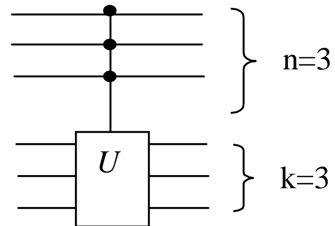


Figure 3- 3 Circuit for the $C^n(U)$ operation when n=3 and k=3

Chapter 4 Špalek's Algorithm

In this chapter, we will briefly review some important aspects of Špalek's algorithm. This chapter follows Høyer and Špalek's paper [HS03]. In the circuits below, we will need the following tools: the quantum fan-out operation, the parallelization method, the quantum Hadamard transformation, and the increment operation.

4.1 Quantum Fan-out Operation

As in classical circuits, fan-out is also an important operation in quantum circuits. One reason is that if two quantum operations commute, then each of them can be performed in parallel on a distinct copy of a qubit, which is produced by applying the fan-out operation.

There is one important difference between the classical circuits and quantum circuit: because of the 'no-cloning' theorem, quantum circuits do not have a naïve fan-out operation that performs

$$|s\rangle|0\rangle^{\otimes n} \rightarrow |s\rangle^{\otimes n+1} \qquad (4.3.1)$$

for a general superposition state $|s\rangle$. However, in [HS03], Špalek using ideas from earlier papers defined a modified quantum fan-out operation that performs

$$|s\rangle\bigotimes_{K=1}^{n}|t_k\rangle \rightarrow |s\rangle\bigotimes_{K=1}^{n}|t_k\oplus s\rangle \qquad (4.3.2)$$

where $|s\rangle$ is the source qubit and there are *n* target qubits $|t_k\rangle$. The effect of (4.3.2) on each computational basis state is the same as (4.3.1), and the effect on the super states is determined by linearity.

4.2 Parallelization Method

Having a model of quantum circuits with unbounded fan-out, we are now able to perform a more general task of applying an arbitrary number of commuting operations in parallel on an individual qubit.

The idea of parallelization comes from the following observation: first, if some operators commute, then they are all diagonal in the same basis. That means, they consist of just phase shifts. Second, we can parallel these multiple phase shifts as following:

1. By applying the fan-out operation, a qubit is duplicated to multiple copies.
2. For each distinct copy apply the commute operation in parallel.
3. The ancilla qubits are initialized to $|0\rangle$, by applying the fan-out operation again, they can be cleaned at the end for reuse.

This method can be extended to multiple qubi ts by copying all target qubits. Other tricks are similar to the one qubit case.

4.3 Quantum Hadamard Transform

The quantum Fourier Transform (QFT) is one of the important tools used in many quantum algorithms. The main trick used in Špalek's algorithm is replacing QFT by the Hadamard transform. Špalek proved the equivalence of using these two transforms for his algorithm.

The Hadamard transform H_n on n qubits is the following operation (written in the computational basis):

$$H_n = 1/2^{n/2} \sum_{j=0}^{2n-1} |y\rangle \sum_{j=0}^{2n-1} (-1)^{y \cdot x} \langle x|,$$

where y·x is the bitwise scalar product. A useful property of the Hadamard transformation is $H_n = H^{\otimes n}$. Figure 4-1 shows an example of $H^{\otimes 2}$. When each qubits is initialed to $|0\rangle$,

after applying the Hadamard operations, the output will be $(|00\rangle + |01\rangle + |10\rangle + |11\rangle) / 2$.

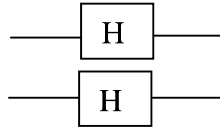


Figure 4- 1 The Hadamard transform $H^{\otimes 2}$ on two qubits

4.4 Increment Operation

Špalek defined an increment operation P on n qubits to be an operation that maps each computational basis state $|x\rangle$ to $|x+1 \bmod 2^n\rangle$. He also proved that P is diagonal in the Fourier basis. So, in this basis, these increment operations can be implemented in parallel by a depth one quantum circuit.

Let $D = FPF^{\dagger}$, that is, the increment operation in the Fourier basis. Define a rotation operator about the z-axis by angle θ by $R_z(\theta) = |0\rangle\langle 0| + e^{\theta i}|1\rangle\langle 1|$. Then for every $k \in \{1, 2, .., n\}$, $D_k = R_z(\pi / 2^{n-k}) \otimes D_{k-1}$. The 0-qubit operation D_0 is considered to be '1'.

4.5 Value Gate

Having all these tools, a quantum circuit for value gate with unbounded fan-out can be constructed as shown in Figure 4-2.

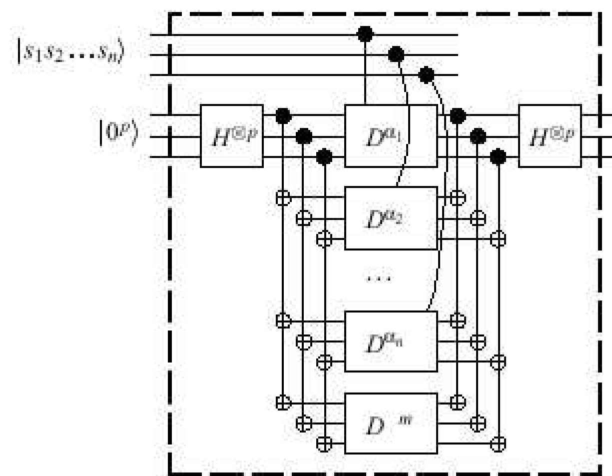


Figure 4 - 2 Quantum circuit for value gate

Chapter 5 Design and Implementation

One of the most challenging parts in simulating this algorithm is to understand the substantial mathematics involved. It took me a long time to figure out the matrix for each quantum gate in the circuit. In this part we demonstrate the matrix representation of the circuit.

Then, in Section 5.2, we show a naïve implementation for classically simulating Špalek's algorithm. This is a good start to understand from the view of implementation that how the algorithm works, and it is also a necessary step to explore much more efficient implementation ideas. However, a naïve implementation needs both exponential space and time complexity. It is infeasible to calculate more than five qubits even using a super machine given this implementation.

In Section 5.3, we introduce a very efficient idea for implementation, which reduces both the time complexity and the space complexity from exponential to polynomial, thus, is feasible on any qubit case.

5.1 Matrix Representation

From the view of implementation, we can divide the circuit, illustrated in Figure 4-1, into 11 layers, as shown in Figure 5-1. From left to right these layers are: Hadamard layer, Permutation layer(P1), Fan-out layer, Permutation layer(P1´), Permutation layer(P2), Increment layer, Permutation layer(P2´), Permutation layer(P1), Fan-out layer, Permutation layer(P1´), and at last, Hadamard layer.

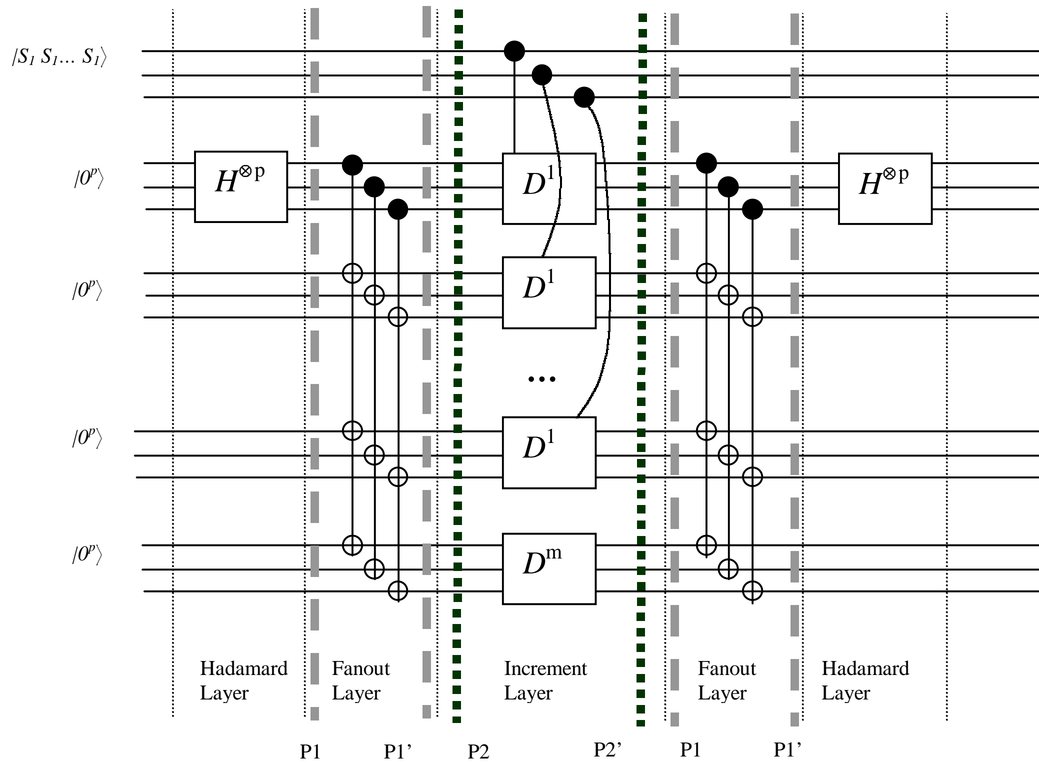


Figure 5- 1 Layers of the circuit for value gate

Among these 11 layers, there are four types of different layers, shown as Hadamard layer, Permutation layer, Fan-out layer, and Increment layer respectively. Other layers just repeat these layers. Each layer can be represented by a matrix, which is the tensor product of sub-matrices. The entire circuit can be represented by a matrix too, which is the result of multiplying the matrices for each layer.

5.1.1 *Matrix for the Hadamard layer*

The matrix representation for the Hadamard layer, denoted as *H_layer*, is very straightforward:

$$\text{H_layer} = \underbrace{I \otimes \ldots \otimes I}_{n} \otimes H^{\otimes p} \otimes \underbrace{I \otimes \ldots \otimes I}_{n.p}$$

where, n is the length of the input, $p = 1 + \lceil \log_2 (1 + n) \rceil$.

5.1.2 Matrix for the Permutation layer

A permutation operator changes the order of the rows of an input vector. Given an input vector *v1*, after applying a permutation matrix, we get another vector *v2*, which has the same entries as *v1*, but in a different order, as shown in Figure 5-2.

$$\underbrace{\begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} a_1 \\ a_2 \\ \cdot\cdot \\ a_n \end{pmatrix}}_{v1} = \underbrace{\begin{pmatrix} a_5 \\ a_1 \\ \cdot\cdot \\ a_7 \end{pmatrix}}_{v2}$$

Figure 5- 2 Permutation operator

Suppose the entries in vector *v1* are ordered as $x = (1, 2, \ldots, n)$, and we want these entries to be ordered as $y = (y_1, y_2, \ldots, y_n)$. The corresponding permutation matrix, *P*, would be:

$$\begin{cases} P[x_i][y_i] = 1 & \text{when } x_i \in x,\ y_i \in y,\ i \in Z, \text{ and } 1 \le i \le n \\ P[x_i][j] = 0 & \text{when } x_i \in x,\ y_i \in y,\ j \ne y_i,\ i, j \in Z, \text{ and } 1 \le i, j \le n \end{cases}$$

In circuit shown in Figure 5-1, six permutation matrices are used. Before applying the Fan-out layer, we first apply the permutation operator *P1* to change the vector's order so that it is suitable for applying Fan-out layer. Then, we use another permutation operator *P1´* to change the order back. When applying the Increment layer, we do the same thing. However, because of the requirements on the order changing are different from the Fan-out layer, we use permutation operator *P2* to change the order, and use *P2´* to change the order back. Here, *P1´* and *P2´* are the transpose matrices of *P1* and *P2* respectively.

5.1.3 Matrix for the Fan-out layer

Figuring out a matrix representation for Fan-out layer directly from Figure 5-1 is difficult. Since we already have the Permutation layer, we can make things more straightforward by making some changes, as illustrated in Figure 5-3.

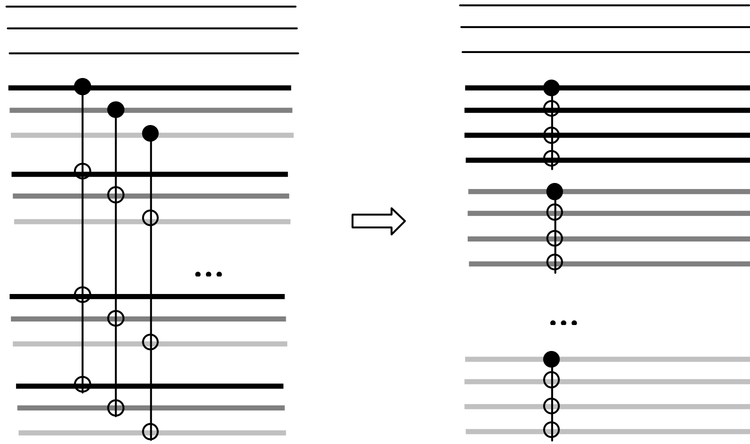


Figure 5- 3 Fan-out layer transformation

Consider the right part of the above figure, we have a matrix representation, *F_layer*, for the Fan-out layer as:

$$\text{F_layer} = \underbrace{I \otimes \ldots \otimes I}_{n} \otimes \underbrace{f \otimes \ldots \otimes f}_{p}$$

where, n is the length of the input, $p = 1 + \lceil \log_2 (1 + n) \rceil$, f is the matrix for the fan-out gate with n target qubits.

According to Špalek's algorithm, a fan-out operation, f, with source qubits $|s\rangle$ and n target qubits $|t_k\rangle$ is defined as:

$$|s\rangle \bigotimes_{K=1}^{n} |t_k\rangle \longrightarrow |s\rangle \bigotimes_{K=1}^{n} |t_k \oplus s\rangle .$$

Thus, we have a matrix for f as shown in Figure 5-4.

$$\begin{bmatrix}
1 & 0 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & 1 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & 0 & 1 & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 1 & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 0 & 0 & \ldots & \ldots & \ldots & 1 \\
0 & \ldots & \ldots & \ldots & 0 & 0 & \ldots & \ldots & 1 & \ldots \\
0 & \ldots & \ldots & \ldots & 0 & 0 & \ldots & 1 & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 0 & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 0 & 1 & \ldots & \ldots & \ldots & 0
\end{bmatrix}
\begin{matrix}
\left.\vphantom{\begin{matrix}0\\0\\0\\0\\0\end{matrix}}\right\} 2^n \\
\left.\vphantom{\begin{matrix}0\\0\\0\\0\\0\end{matrix}}\right\} 2^n
\end{matrix}$$

Figure 5- 4 Matrix for *f* operator

5.1.4 Matrix for the Increment layer

We can use the same method as for the Fan-out layer to make some changes to the Increment layer, and so can make things easier.

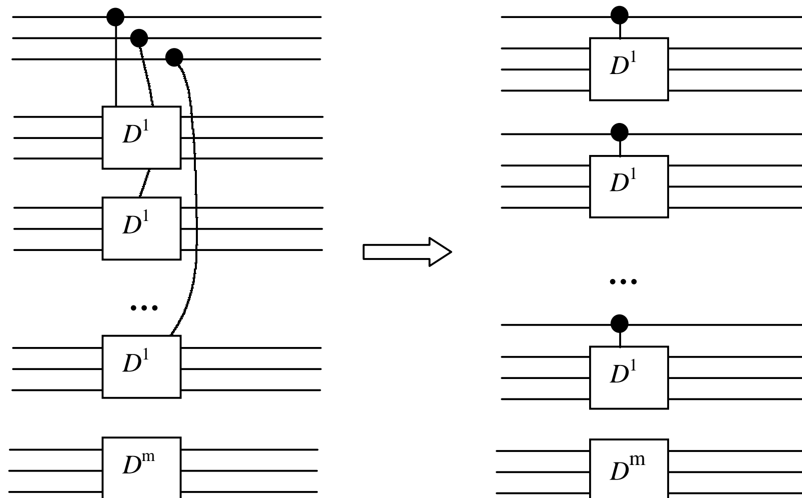


Figure 5- 5 Increment layer transformation

As Figure 5-5 shows, by applying Permutation layer we can make the above transformation. We then have a matrix representation for the right part of the figure. Suppose we use *I_layer* to represent this matrix, then:

$$\text{I_layer} = \underbrace{c_d \otimes \ldots \otimes c_d}_{n} \otimes d$$

where, n is the length of the input, $p = 1 + \lceil \log_2 (1 + n) \rceil$, d is the matrix for increment gate with p qubits, c_d is the matrix for controlled increment gate on p target qubits.

The increment operator d is explained in detail in [HS03]. We now examine the matrix representation for c_d, which is a 'controlled-U' operation, also called a $C^n(U)$ operation. In general, we cannot find a simple matrix representation for a $C^n(U)$ operation, fortunately, in our case we can limit our considerations to $C^1(U)$, where the unitary operator U is replaced by d. In this case, we have a matrix representation for c_d as shown in Figure 5-6.

$$\left[\begin{array}{ccccc|ccccc}
1 & 0 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & 1 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & 0 & 1 & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 1 & 0 & \ldots & \ldots & \ldots & 0 \\
0 & \ldots & \ldots & \ldots & 0 & & & & & \\
0 & \ldots & \ldots & \ldots & 0 & & & & & \\
0 & \ldots & \ldots & \ldots & 0 & & & d & & \\
0 & \ldots & \ldots & \ldots & 0 & & & & & \\
0 & \ldots & \ldots & \ldots & 0 & & & & &
\end{array}\right]
\begin{array}{l} \left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} 2^n \\ \left.\vphantom{\begin{array}{c}1\\1\\1\\1\\1\end{array}}\right\} 2^n \end{array}$$

(column blocks: 2^n, 2^n)

Figure 5- 6 Matrix for $C^1(d)$ operator

5.1.5 Matrix for the entire circuit

Having the matrix for each layer, we now turn to the matrix representation for the entire value gate circuit. A matrix (M) for value gate is made from the product of each layer, which is:

$$M = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1 \cdot H_layer$$

5.2 A Naïve Implementation

In this section we want to introduce our initial implementation. This implementation is the basis for us to explore a much more efficient implementation idea. A natural way do to the simulation is to generate and store each matrix. Then, we can do calculations on them, such as addition, multiplication, tensor-product, etc. Further, we can generate quantum gates using these matrices.

5.2.1 Class organization

In my implementation, four classes were used: *Complex number*, *Matrix*, *Gate* and *Value gate*. In the *Complex number*, we define a complex number and its operations. In the *Matrix*, the notion of matrix and operations on it are defined. The *Gate* class works like a gate factory, we define and generate base quantum gates that are needed by each layer. At last, the *Value gate* class defines a value gate using each layer that was described in Section 5.1, and can be used to evaluate the entire circuit. A class diagram is show in Figure 5-7.

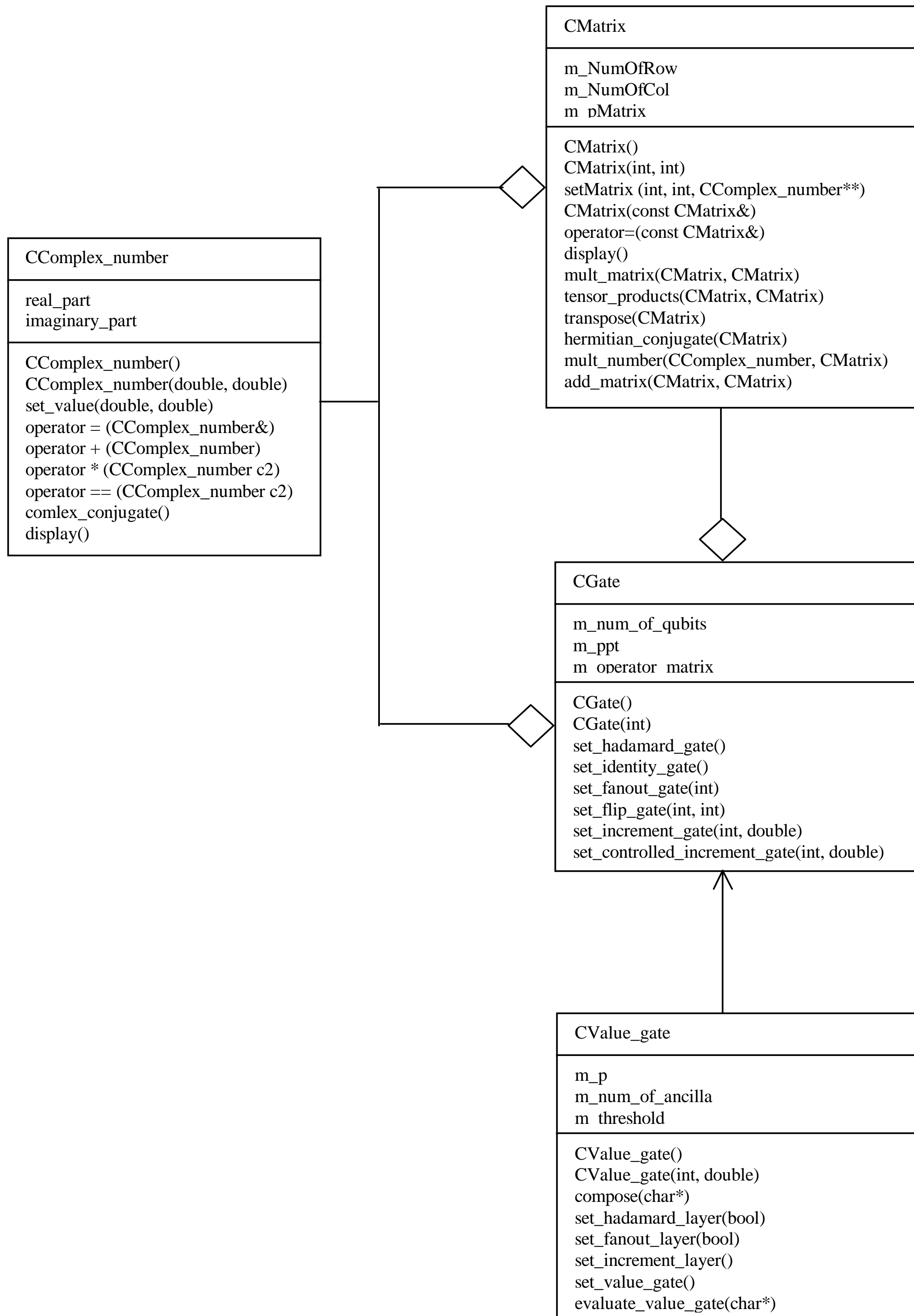


Figure 5- 7 Class diagram

5.2.2 Limitation of the naïve implementation

As we shown in Section 5.1, to evaluate the value gate, we need to generate 11 matrices for each layer, and calculate their product from right to left. Notice that the size for each such matrix is: $2^{n+(n+1)logn}$ *by* $2^{n+(n+1)logn}$, thus, both the space complexity and time complexity are $O(2^{nlogn})$. Storing and computing on such huge matrices is very machine intensive. For example in the five qubits case, we will need at least 8G bytes memories. So the exponential time and space complexities make using this program on even small inputs impractical.

5.3 A Very Efficient Implementation

In this part, we introduce an idea for implementing Špalek's algorithm that uses polynomial time and space complexity. We begin with a rough sketch of the idea. Then we explain how we do the implementation in detail. Finally, we show our way of introducing error scheme to the simulation.

5.3.1 Sufficient for reducing computational complexity

Normally, simulating a quantum algorithm is very expensive because of the need to calculate with exponential sized matrices. Given a quantum circuit, one may or may not able to find a way to reduce the complexity to polynomial; it depends on the specific algorithm involved. Our idea is based on three ideas:

1. For result matrix M (see Section 5.1.5), we are only interested in one element. See Section 5.3.1.1 for details.
2. In each matrix that represents a layer, there are at most $O(n)$ non-zero elements in each row/column, which is explained in Section 5.3.1.2.

3. Every row/column in any layer matrix can be computed in polynomial time. This will be discussed in Section 5.3.1.3.

5.3.1.1 Determining which element is needed

Recall in Section 5.1, we calculated the matrix M as

$$M = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1 \cdot H_layer$$

we can calculate the value-gate value val as:

$$val = \underbrace{\begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}}_{V} \underbrace{\begin{pmatrix} m_{11} & m_{12} & \ldots & m_{1n} \\ m_{21} & m_{22} & \ldots & m_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ m_{n1} & m_{n2} & \ldots & m_{nn} \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} v_1 \\ v_2 \\ \cdots \\ v_n \end{pmatrix}}_{V'}$$

In vector V, we have:

$$v_i = \begin{cases} 1, & \text{if } i = S \cdot 2^{(n+1) \cdot p}, \quad S, n, p \text{ as in Figure 5.1} \\ 0, & \text{otherwise} \end{cases}$$

From here we can see that in order to calculate the value of val, we do not need to calculate the entire matrix M, knowing the element $M[S \cdot 2^{(n+1) \cdot p}, S \cdot 2^{(n+1) \cdot p}]$ is sufficient for us.

5.3.1.2 Reducing the size of problem from exponential to polynomial

As we discussed earlier, we have

$$M = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1 \cdot H_layer,$$

suppose that we have already calculated

$$X = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1,$$

and that we want to go ahead to figure out the value of $M[S \cdot 2^{(n+1) \cdot p},\ S \cdot 2^{(n+1) \cdot p}]$. In this case, we are only interested in the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *X*, thus the problem is reduced to two sub-problems:

a) Sub-problem SP1: how to efficiently calculate the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of X

b) Sub-problem SP2: once we get calculate the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of X, how to efficiently calculate $M[S \cdot 2^{(n+1) \cdot p},\ S \cdot 2^{(n+1) \cdot p}]$.

Later we'll see that SP2 is simpler than SP1 and can be solved in the same way as solving SP1. Now we start discussing how to solve SP1:

a). Solving SP1

We now describe a recursive way of solving SP1. Suppose we already have the result of

$$Y = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer$$

then we have:

$$X = Y \cdot P1.$$

Note that we do not need to calculate the entire matrix *Y* to solve SP1 (i.e., calculate the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of X), having value of ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *Y* is enough to calculate the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *X*.

Recursively applying this idea, we have a method to solve SP1 as follows:

1). Calculate ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *Z*, where

$$Z = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1'$$

2). Step 1) can be calculated by calculating ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *Q*, where

$$Q = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2$$

3). Step 2) can be calculated by calculating ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of R, where

$$R = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer$$

…

n) We can start from ($S \cdot 2^{(n+1) \cdot p}$)'*th* row of *H_layer* and move forward to finally compute 1)

Remark 1: In this implementation, for intermediate results, we only save the ($S \cdot 2^{(n+1) \cdot p}$)'*th* row for each layer matrix and for the final matrix V.

So far, we reduced the intermediate result size from the entire matrix to one row. Yet this is not sufficient to reduce time and space complexity to polynomial, since there are exponential many of elements in each row.

Before we go further, let's first study the layer matrix a little more. From our matrix definition for each layer, we can observe the following property of layer matrices:

Proposition 1:

- There are $2^p = O(n)$ non-zero elements per row/column in the matrix for Hadamard layer.
- There is exactly one non-zero element per row/column in the matrix for fan-out layer.
- There is exactly one non-zero element per row/column in the matrix for increment layer.

- There is exactly one non-zero element per row/column in the matrix for permutation layer.

Based on the above observations, we have the following lemma:

Lemma 1: In the value gate matrix X, where

$$X = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1$$

there are at most $O(n)$ non-zero elements in $(S \cdot 2^{(n+1) \cdot p})$*'th* row.

Proof: We calculate X from left to right:

- First, row $(S \cdot 2^{(n+1) \cdot p})$ of *H-Layer* has $O(n)$ non-zero elements,
- Because $P1'$ has only one nonzero element in each column, then row $(S \cdot 2^{(n+1) \cdot p})$ of $H\text{-}Layer \cdot P1'$ has at most $O(n)$ non-zero elements
- For the same reason, we can continue compute until reach

  $X = H_layer \cdot P1' \cdot F_layer \cdot P1 \cdot P2' \cdot I_layer \cdot P2 \cdot P1' \cdot F_layer \cdot P1$ ■

Thus, we note that:

Remark 2: In intermediate results of the value gate, i.e., the $(S \cdot 2^{(n+1) \cdot p})$'th row of the matrix, there are at most $O(n)$ non-zero elements, while the total number of elements in this row is $O(2^{nlogn})$.

At this point, we have reduced the problem of calculating $O(2^{nlogn})$ elements to a problem of calculating $O(n)$ elements. Now we give an efficient way of calculating these $O(n)$ elements in polynomial time and space.

Consider the following scenario: we have already got the row $r=(r_1,r_2, .., r_n)$, in which r is sparse and only r_i, r_j and r_k are non-zero elements. To multiply the next matrix:

$$next_row = \begin{bmatrix} \ldots r_i \ldots r_j \ldots r_k \ldots \end{bmatrix} \begin{pmatrix} \ldots \\ p_{i1} \; p_{i2} \cdots p_{in} \\ \ldots \\ p_{j1} \; p_{j2} \cdots p_{jn} \\ \ldots \\ p_{k1} \; p_{k2} \cdots p_{kn} \\ \ldots \end{pmatrix}$$

r P

If r is sparse, and r_i, r_j and r_k are non-zero elements, then what will happen during the multiplication? In this case, only the row *i* or *j* or *k* of *P* will multiply against a non-zero value (r_i, r_j and r_k respectively), other rows will simply multiply a ‘0’ from *r*.

What this means is that we do not even need to calculate all rows in *P*, it is sufficient to calculate only rows that correspond to a non-zero element in *r* (in this case, we only need to calculate *i, j* and *k* rows of *P*).

As we already know from Remark 2, *r* has at most *O(n)* non-zero elements, so we need to calculate at most *O(n)* rows.

Because there are only *O(n)* non-zero elements in both the vector and the row of the layer matrix, we can effectively save a vector by a linked-list to achieve polynomial complexity elements.

Now, we are only one step away from getting a polynomial time algorithm. We now explore how to effectively save a layer matrix and how to compute a particular row of it.

From proposition 1, we know that there are at most *O(n)* non-zero elements in a row of any layer matrix. If we can figure out an effective way to calculate a row of a layer matrix, we will be able to find a polynomial value-gate algorithm.

5.3.1.3 Getting a row of a layer matrix

Now we describe the idea of saving the layer and getting one row of it in polynomial time. We discuss two cases: a layer matrix created by tensor product (*H_layer, I_layer, F_layer*) and a permutation layer matrix (*P* layers).

- Case 1: Tensor product matrix

If a matrix is created as the tensor products of sub-matrices, it is very cheap to only save the base matrices that participate in the tensor product. This needs at most polynomial space. The question is how to effectively compute a row as required above?

First, let's study the relationship between rows in the base matrix and the rows in a layer matrix. Take the following simple tensor product as an example:

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \otimes \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00} \cdot b_{00} & a_{00} \cdot b_{01} & a_{01} \cdot b_{00} & a_{01} \cdot b_{01} \\ a_{00} \cdot b_{10} & a_{00} \cdot b_{11} & a_{01} \cdot b_{10} & a_{01} \cdot b_{11} \\ a_{10} \cdot b_{00} & a_{10} \cdot b_{01} & a_{11} \cdot b_{00} & a_{11} \cdot b_{01} \\ a_{10} \cdot b_{10} & a_{10} \cdot b_{11} & a_{11} \cdot b_{10} & a_{11} \cdot b_{11} \end{bmatrix} \begin{matrix} \text{Row 00} \\ \text{Row 01} \\ \text{Row 10} \\ \text{Row 11} \end{matrix}$$

We see that

- row 00 comes from row 0 of 'a' and row 0 of 'b'
- row 01 comes from row 0 of 'a' and row 1 of 'b'
- row 10 comes from row 1 of 'a' and row 0 of 'b'
- row 11 comes from row 1 of 'a' and row 1 of 'b'

In this simple case, if a layer matrix is the tensor product of n (2X2) base matrices $b_1, b_2, \ldots, b_n$, then the row $r_1 r_2 \ldots r_n$, where $r_i \in [0..1]$, can be computed from row $r_1, r_2, \ldots, r_n$ of matrix $b_1, b_2, \ldots, b_n$ respectively. Using a linked-list and a sparse matrix, this can be done in polynomial space.

A similar idea is used for more complex base matrices. For example, if a layer is the tensor product of a (2X2) matrix a and a (4X4) matrix b, then line $r_1 r_2 r_3$ of the layer results from the r_1 row of a and the $r_2 r_3$ row of b.

- Case 2: Permutation matrix

  Please see Section 5.1.2 for more information. This can be done in polynomial time.

Remark 3: The intermediate results of value gate, i.e., the $(S \cdot 2^{(n+1) \cdot p})$'th row of the matrix, can be calculated in $O(n)$ time and space complexity.

b) Solving SP2

Once we calculate SP1 as a vector $\boldsymbol{r}$, the situation is as following:

$$final_row = \underbrace{\begin{bmatrix} \ldots r_i \ldots r_j \ldots r_k \ldots \end{bmatrix}}_{r} \underbrace{\begin{pmatrix} \ldots \\ h_{i1} \; h_{i2} \ldots h_{in} \\ \ldots \\ h_{j1} \; h_{j2} \ldots h_{jn} \\ \ldots \\ h_{k1} \; h_{k2} \ldots h_{kn} \\ \ldots \end{pmatrix}}_{H}$$

And now our final goal is to get the $(S \cdot 2^{(n+1) \cdot p})$'th element of *final_row*.

The only values in the *H_layer* that contribute to *final_row*$[S \cdot 2^{(n+1) \cdot p}]$ are column $(S \cdot 2^{(n+1) \cdot p})$, see below:

$$final_row(S \cdot 2^{(n+1) \cdot p}) = \sum^{x = i,j,k,\ldots} (r_x \cdot h_{x,\,(S \cdot 2(n+1) \cdot p)})$$

In the *H_layer*, there are $O(n)$ non-zero elements in each row, if we use the algorithm in SP1 to create the row, then each row takes $O(n)$ time to create, the cost of finding its $(S \cdot 2^{(n+1) \cdot p})$'th element is also $O(n)$, thus the complexity of finding all h_x is $O(n^2)$. Computing *final_row*$[S \cdot 2^{(n+1) \cdot p}]$ is then trivial and costs $O(n)$. Thus the total cost is $O(n^2) + O(n) = O(n^2)$.

Remark 4: $M[S \cdot 2^{(n+1) \cdot p},\ S \cdot 2^{(n+1) \cdot p}]$ can be calculated in $O(n^2)$ time and space complexity.

5.3.2 Implementation detail

5.3.2.1 Saving each row of the layer matrix

A natural way to save a sparse row of any layer matrix is to use a linked-list. We used the following data structure to represent one node of this linked-list:

```
struct node
{     char                    column_index[Max_NP+1];
      CComplex_number         value;
      node*                   next;
};
```

Normally, we use the data type 'integer' to represent the index of a row/column, but as you can see, we are using a string instead in this implementation. In Špalek's algorithm, given *n* bits input, there will be *O(nlogn)* lines in the circuit, thus, we will need a vector of length $2^{O(nlogn)}$ to represent this quantum state. However, 32 bits is not enough to store the result as *n* gets large. This is reason for using a '0/1' string for index representation.

5.3.2.2 Computing and saving the layer's matrix

Now the question is how to efficiently save and compute the layer matrices, i.e., *H_layer, F_layer, I_layer, Ps.*

- The matrix for Hadamard layer: it is the tensor product of small matrices. We can easily compute a row from its base matrices. Saving the base matrices is enough.
- The matrix for fan-out layer: saving the base matrices is enough to compute a row.
- The matrix for increment layer: saving the base matrices is enough to compute a row.
- The matrix for permutation layer: we can easily compute a row by its permutation table. Saving the permutation table is enough.

For the Hadamard, the Fan-out and the Incremental matrices, we can save their base matrices, and use the algorithm described in Section 5.3.1.3 to compute any row. The pseudo-code is listed below:

Function get_matrix_row(Matrix M, char *r)

Input: A char* for row

A matrix **M**, represented using its base matrices

Output: row r of matrix M, saved in a linked-list for its non-zero elements

Variables:

struct node *p, *q, *x, *y;

char subrow[MAX_ROWS][MAX_LEN];

int i, num;

Begin

num = M->num_base_matrices;

Based on the sizes of M's base matrices, split the input string r into subrow corresponding to the log size of each base matrix

/*********************************

For example, if there are two base matrices b1 and b2, b1 is a (4X4) matrix and b2 is a (8X8) matrix, then the input "01101" should be split into subrow[0]="01" and subrow[1]="101"

*********************************/

p = get_row_of_base_matrix(M->base_matrix[0], subrow[0]);

for (i=1; i<num; i++)

Begin

q = get_row_of_base_matrix(M->base_matrix[i], subrow[i]);

x = NULL;

while (p != NULL)

```
Begin
  allocate y;
  while (q != NULL)
  Begin
    y ->row = strcat(p->row, q->row);
    y ->value = p->value * q->value;
    link node y into x;
    q = q->next;
  End
  p = p->next;
End
p = x;
End
return p;
End
```

5.3.2.3 Computing a intermediate row

Computing an intermediate row can be efficiently implemented using a linked-list. The pseudo-code for multiplication is:

Function row_calculation(Row_pointer r, Matrix M)

Input: a row **r** of non-zero elements, represented using a linked-list

A matrix **M**, represented using either its base matrix (it **M** is generated by tensor product , or the permutation map, if **M** s a permutation matrix)

Output: a new row of $\mathbf{r} \cdot \mathbf{P}$

Variables:

```
struct node *p, *q, *x[MAX_ROWS], result;
char *row;
int i, num=0;
double v;
boolean finish;
```

Begin

```
/******* Step 1: Compute all output lines into x[]*******/
    p = r;
    while ( p != NULL )
    Begin
        v = p->value;
        row = p->row;

        x[num] = get_matrix_row (M, row);
        q = x[num];
        while (q != NULL)
        Begin
            q->value *= v;
            q = q->next;
```

```
    End
    num++;
    p = p->next;
End
/******* Step 2: merge x[] into one single row *******/
finish = false;
while ( !finish )
Begin
  find k such that x[k] has the smallest column id;
  allocate new p,
  p->value = 0.0;
  p->column = x[k]->column;
  insert p into result;
  for (i=0; i<num; i++)
  Begin
     if (x[i]->column == p->column)
     Begin
         p->value += x[i]->value;
         x[i] = x[i] ->next;
     End
  End
  finish = (all x[i]==NULL)
End
```

return result;

End

5.3.2.4 Class organization

A class diagram for this project is shown in Figure 5-8. The *Complex_number* class is used to define a complex number and its operation. *Base_matrix* is an abstract class, and five other classes are derived from it. We define five base quantum gates described in Section 5.3.2.3 as classes *Fan-out_matrix*, *Identity_matrix*, *Hadamard_matrix*, *Controlled_D_matrix*, *and D_matrix* respectively. Each layer matrix is made from either the tensor product of these base matrices or using permutations; we use the *TensorProduct_layer* class and *Permutation_layer* class to implement these ideas. The *Multiplication* class is used for producing intermediate rows. Finally, the *Value_gate* class is used to construct a quantum value gate circuit and evaluate it.

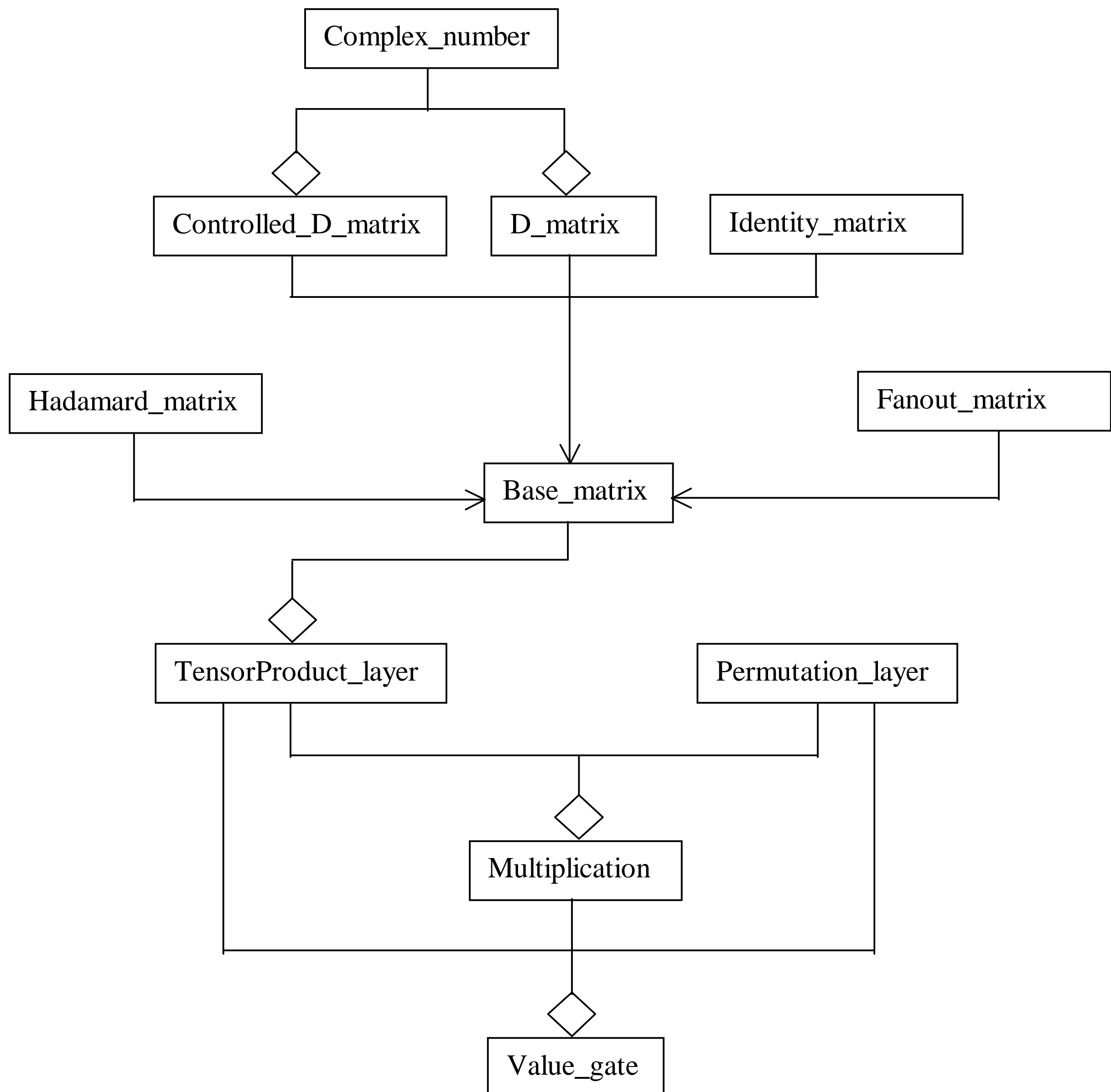


Figure 5- 8 Class diagram

5.4 Error scheme

Špalek's algorithm assumes that we can perform certain rotation operators to arbitrary accuracy, but this is not the case in practice. To this point, our implementation is

based on the ideal case. To simulate a more realistic criterion, we add an error scheme to our system.

In Špalek's algorithm, a rotation operator about the z-axis by angle θ is defined by

$$R_z(\theta) = |0\rangle\langle 0| + e^{\theta i}|1\rangle\langle 1|,$$

which is

$$R_z(\theta) = |0\rangle\langle 0| + (\cos\theta + i\sin\theta)\, |1\rangle\langle 1|,$$

In increment layer (*I_layer*) each increment operator (*D*) is made from:

$D_k = R_z(\pi/2^{n-k}) \otimes D_{k-1}$, where k is the number of qubits, $D_0 = 1$.

Suppose we have an error ε, we add errors in two ways:

1) Add a fix error ε to each rotation operator, that is

$$R_z(\theta+\varepsilon) = |0\rangle\langle 0| + (\cos(\theta+\varepsilon) + i\sin(\theta+\varepsilon))\, |1\rangle\langle 1|.$$

Then, D_k is made from the tensor product of the one qubit rotation operators with the same error.

2) For each rotation operator, random select an error ε', which is between $-\varepsilon$ and ε, add ε' to this rotation operator, that is

$$R_z(\theta+\varepsilon') = |0\rangle\langle 0| + (\cos(\theta+\varepsilon') + i\sin(\theta+\varepsilon'))\, |1\rangle\langle 1|.$$

So, we have different error ε' for each rotation operator, thus, D_k is made from the tensor product of the one qubit rotation operators with different errors.

Chapter 6 Tests

There are two purposes for our tests. First, we want to verify the correctness of the simulation program. Second, observing test results can help us to understand the behavior of the simulated algorithm with errors.

6.1 Test Case Design

In an ideal case where there is no error added to the system, the simulation should perform the exact function as a value gate. This gives a way to verify if my program works correctly. Thus, we designed test case 1 as follows:

Test Case 1

String: Arbitrary 0/1 string with arbitrary length.

Threshold: 1) An integer that equals to the number of bits that is 'on' in the input string. 2) An arbitrary integer that does not equal to the number of bits that is 'on' in the input string.

Error: 0

Output: Probability for accepting, which is computed by calculating $|M[S \cdot 2^{(n+1) \cdot p}, S \cdot 2^{(n+1) \cdot p}]|^2$

In practice, we cannot perform certain rotation operations to arbitrary accuracy. To simulate this phenomenon, we need to introduce errors into the system. It is interesting to see how these errors affect the behavior of the Špalek's algorithm.

We designed two test cases for such observations that we call Test Case 2 and 3. Test Case 2 is designed to see how we will lose the ability to give a correct result as the error size increases. Given a Hamming length (the number of bits that is 'on' in the input

string) and a length of input string, we can generate all possible strings. In test case 3 we want to see how the algorithm acts on such strings.

In real quantum devices, it is more likely that for each rotation operation, a different error will occur. To see the effects of this kind of factor, in both Test Case 2 and Test Case 3, we tested two cases: first, set the error for each rotation operator the same; second, set the error for each rotation operator randomly in some range.

Test Case 2:

For each run we have:

String: A fixed '0/1' string.

Threshold: A fixed integer either from 1) an integer which equals to the number of bits that is 'on' in the input string, or 2) an arbitrary integer which does not equal to the number of bits that is 'on' in the input string.

Error: A real number, ε. In the case of setting the same error to each rotation operator, ε is applied. In the case of setting different error to each rotation operator, a random selected real number, which between -ε and ε is applied.

Same Rotation Error: A fixed Boolean value either from 1) true for setting the same error for each rotation operator, or 2) false for setting different ones.

Output: Probability for accepting, which is computed by calculating $|M[S \cdot 2^{(n+1) \cdot p}, S \cdot 2^{(n+1) \cdot p}]|^2$

We tested 100 runs for each sub-case. Each run represents one point in the test result figure. X-axis represents $i \cdot \varepsilon/100$, where i is a integer and $0 <= i < 100$, that is, 100 equal distance ascend errors from 0 to ε. Y-axis represents the probability for accepting.

Test Case 3:

For each run we have:

Length of String: A fixed integer number, which shows the length of the input string.

Hamming Length of String: A fixed integer number, which shows how many bits of the input string are set to be 'on'.

Threshold: A fixed integer either from 1) an integer that equals to the number of bits that is 'on' in the input string, or 2) an a rbitrary integer that does not equal to the number of bits that is 'on' in the input string.

Error: A fixed real number, say ε. In the case of setting the same error to each rotation operator, ε is applied. In the case of setting different error to each rotation operator, a random selected real number, which between -ε and ε, is applied.

Same Rotation Error: A fixed Boolean value either from 1) true for setting the same error for each rotation operator, or 2) false for setting different ones.

Output: Probability for accepting, which is computed by calculating $|M[S \cdot 2^{(n+1) \cdot p}, S \cdot 2^{(n+1) \cdot p}]|^2$

We tested all possible input strings with the given length and the given Hamming length. The X-axis represents each possible string with the same length and the Hamming length in a lexicographical order. The Y-axis represents the probability for accepting.

6.2 Test Results Sample

As described in Chapter 5, the simulator runs in polynomial time and polynomial space. It is thus feasible to run tests on many qubits. In our tests, we limited our tests to no more than 30 qubits on a PC and 50 qubits on a HP workstation. For the 30 qubits

case, it took about 4.5 seconds for each run on a 600 MHZ PC. In this section, we show some typical test samples.

Sample of Test Case 1 Results

We ran 200 tests with different input strings and different threshold values, where the error was zero for each run. Our tests show that when the threshold equals the number of bits that are 'on' in the input string, we got output '1', otherwise the output w as '0'. This result is the same as what is described in Špalek's algorithm, thus verifying the correctness of our simulator. Table 6-1 lists 20 sample tests of this case.

Input string	**Threshold value**	**Error**	**Output**
111111111100000000001111111111	20	0	1
111111111100000000001111111111	15	0	0
010101010101010101010101010101	15	0	1
010101010101010101010101010101	25	0	0
000000011111110000000101010	10	0	1
000000011111110000000101010	2	0	0
0000000000000000000000001	1	0	1
0000000000000000000000001	2	0	0
11111111111111111111	20	0	1
11111111111111111111	7	0	0
01110111010000000	7	0	1
01110111010000000	20	0	0
10011000001111	7	0	1
10011000001111	2	0	0
10000111000	4	0	1
10000111000	1	0	0
101	2	0	1
101	1	0	0
1	1	0	1
1	0	0	0

Table 6- 1 Test results for test 1

Sample of Test Case 2 Results
Test 2.1

String: 10000111
Threshold: 4
Error: 1
Same Rotation Error: true

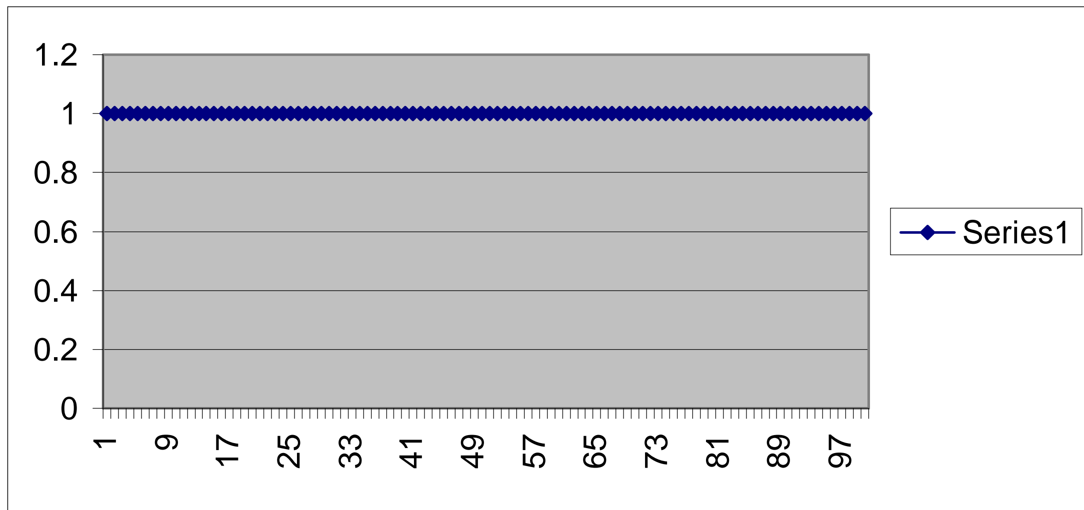


Figure 6- 1 Test result for test 2.1

Test 2.2

String: 10000111
Threshold: 5
Error: 12.5666370614359 (around 4*PI)
Same Rotation Error: true

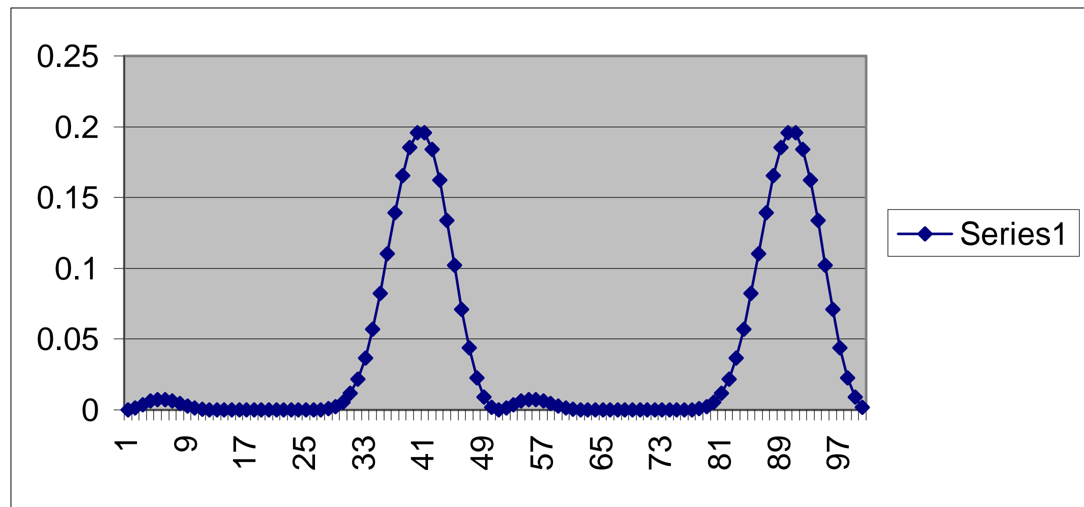


Figure 6- 2 Test result for test 2.2

Test 2.3

String: 111110000011111
Threshold: 12
Error: 12.5666370614359 (around 4*PI)
Same Rotation Error: true

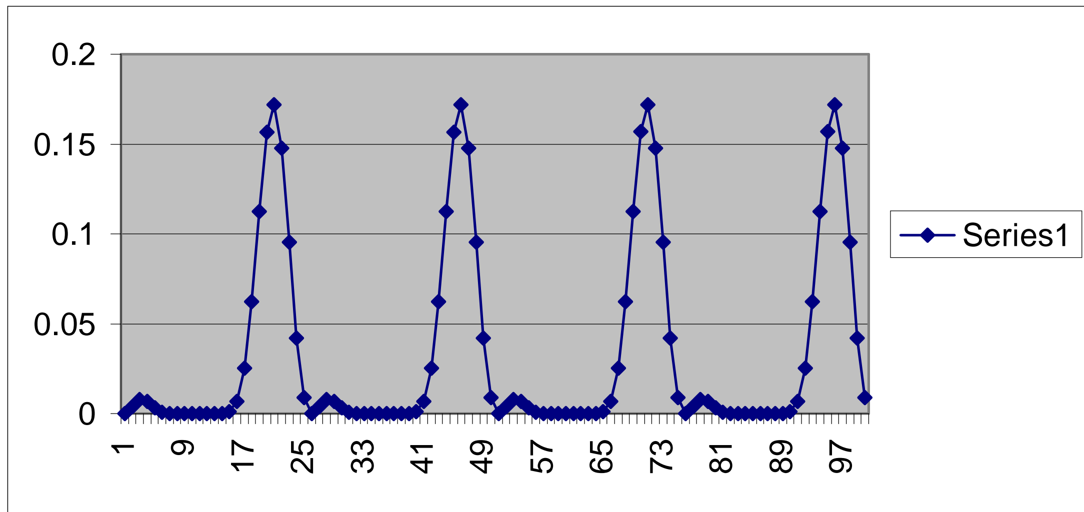


Figure 6- 3 Test result for test 2.3

Test 2.4

String: 10000111
Threshold: 4
Error: 1
Same Rotation Error: false

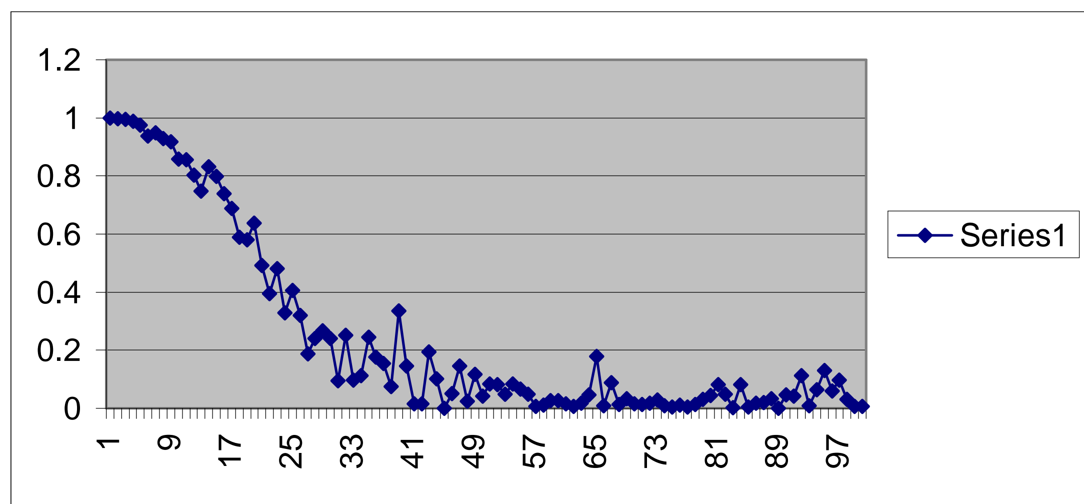


Figure 6- 4 Test result for test 2.4

Test 2.5

String: 000001111100000111110000011111
Threshold: 15
Error: 1
Same Rotation Error: false

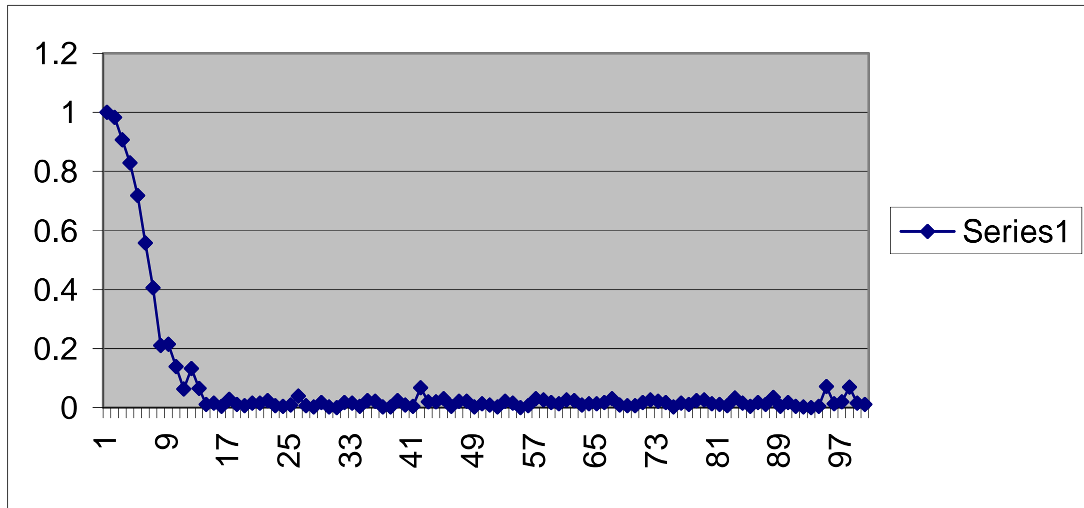


Figure 6- 5 Test result for test 2.5

Test 2.6

String: 10000111
Threshold: 4
Error: 0.1
Same Rotation Error: false

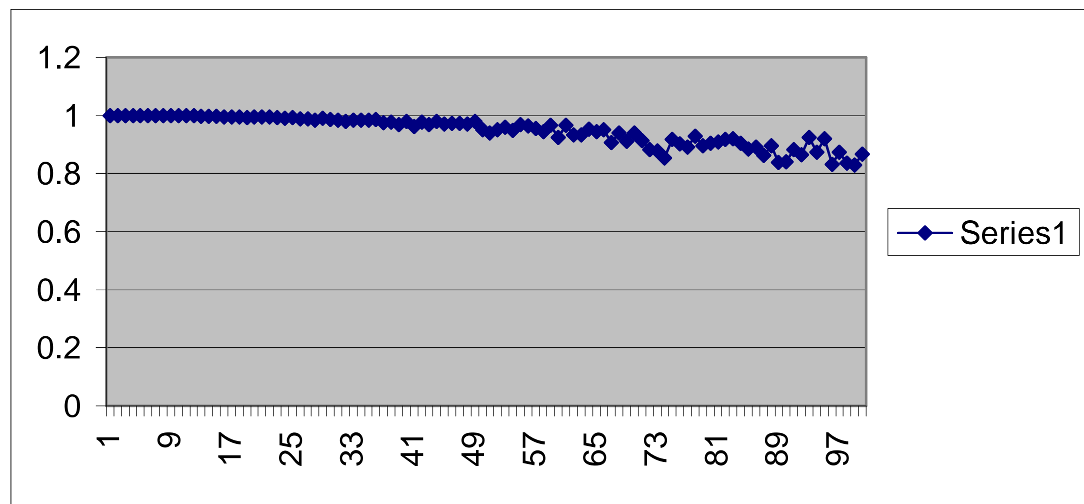


Figure 6- 6 Test result for test 2.6

Test 2.7

String: 0000011111000001111100000011111
Threshold: 15
Error: 0.1
Same Rotation Error: false

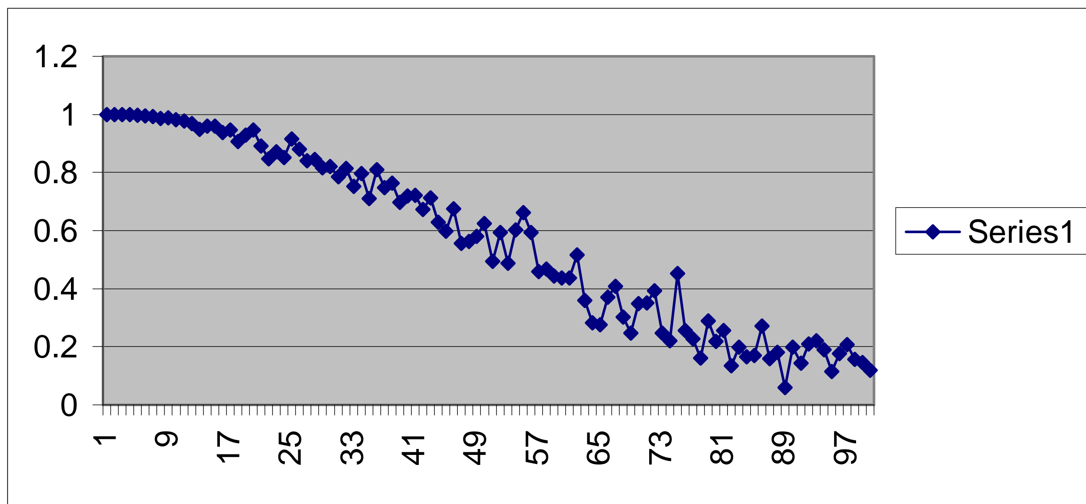


Figure 6- 7 Test result for test 2.7

Test 2.8

String: 10000111
Threshold: 5
Error: 1
Same Rotation Error: false

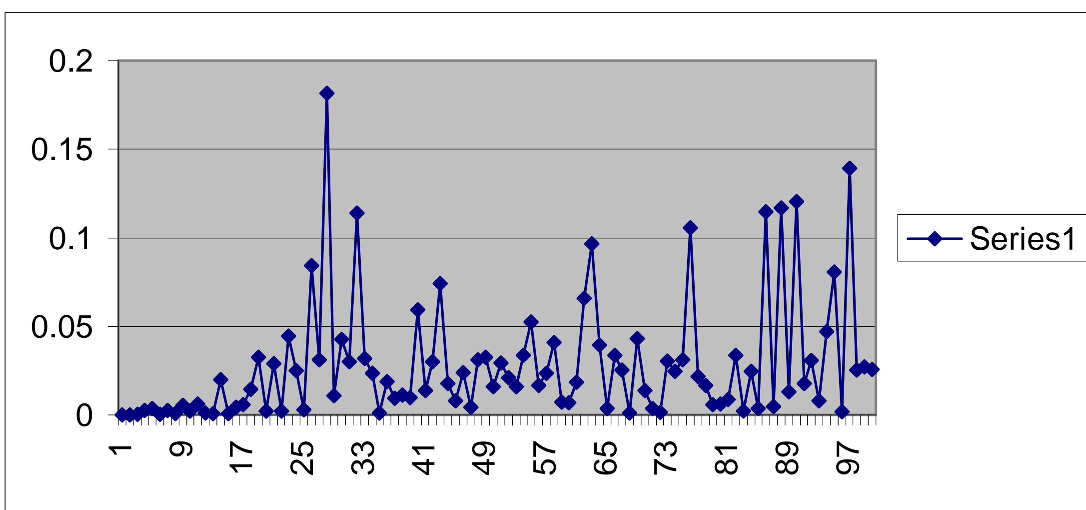


Figure 6- 8 Test result for test 2.8

Test 2.9

String: 000001111100000111110000011111
Threshold: 18
Error: 1
Same Rotation Error: false

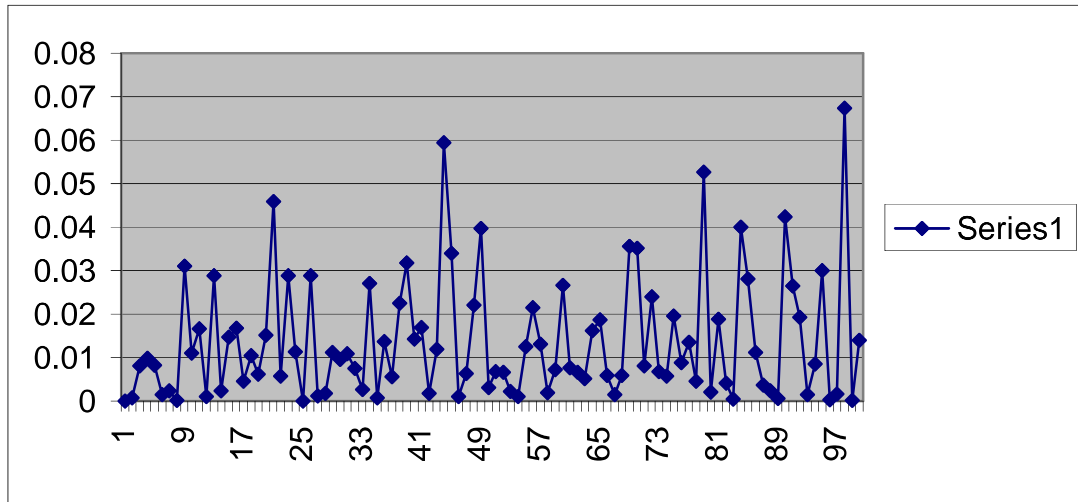


Figure 6- 9 Test result for test 2.9

Sample of Test Case 2 Results
Test 3.1

Length of Input String: 7
Hamming length: 3
Threshold: 3
Error: 1
Same Rotation Error: true

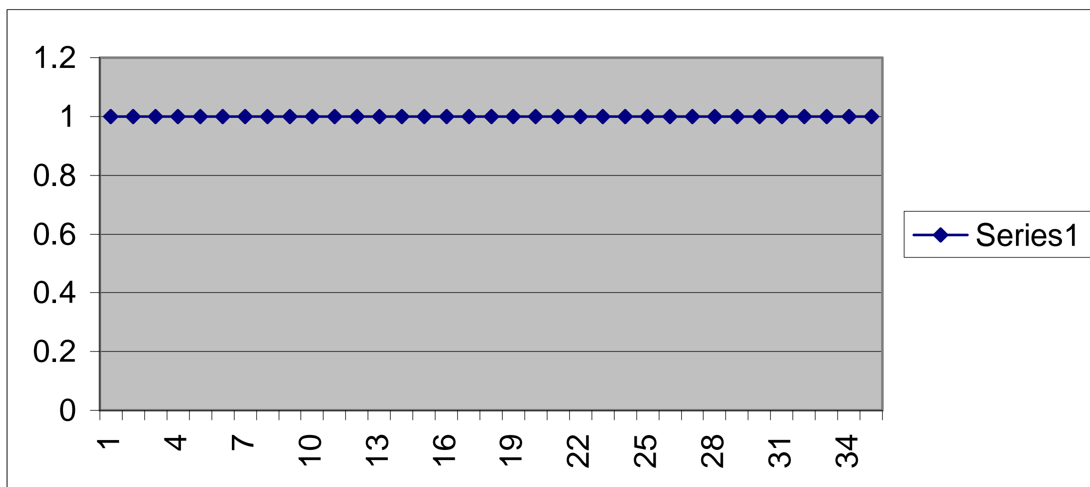


Figure 6- 10 Test result for test 3.1

Test 3.2

Length of Input String: 7
Hamming length: 3
Threshold: 4
Error: 1
Same Rotation Error: true

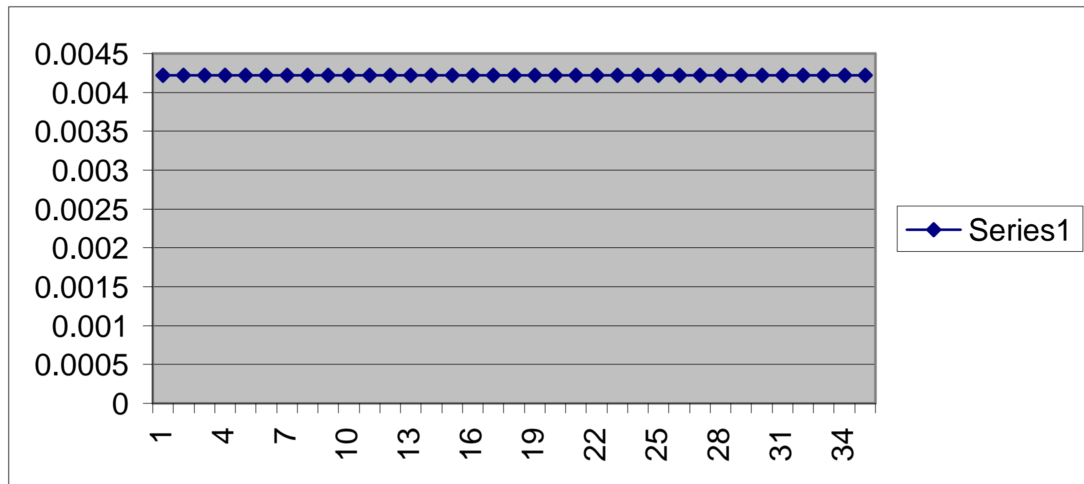


Figure 6- 11 Test result for test 3.2

Test 3.3

Length of Input String: 7
Hamming length: 3
Threshold: 3
Error: 1
Same Rotation Error: false

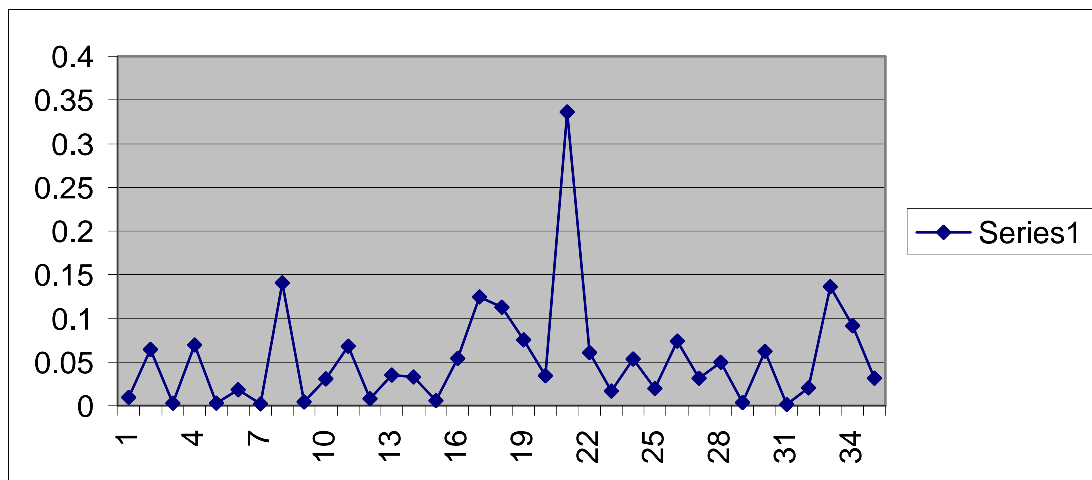


Figure 6- 12 Test result for test 3.3

Test 3.4

Length of Input String: 7
Hamming length: 3
Threshold: 3
Error: 0.05
Same Rotation Error: false

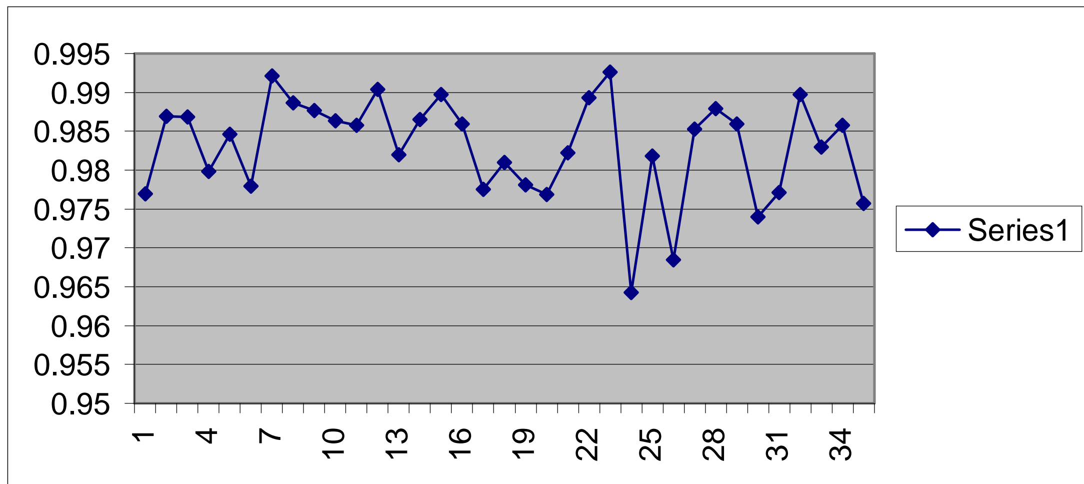


Figure 6- 13 Test result for test 3.4

Test 3.5

Length of Input String: 7
Hamming length: 3
Threshold: 4
Error: 1
Same Rotation Error: false

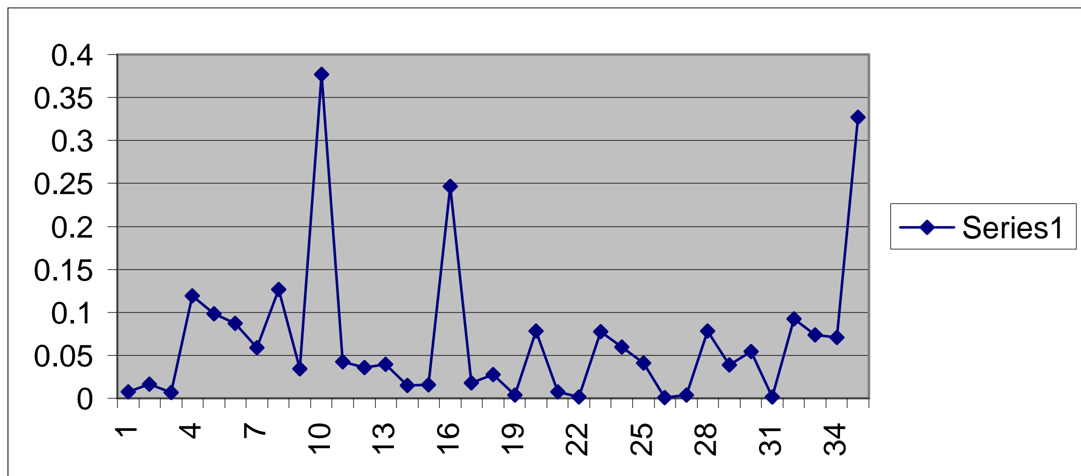


Figure 6- 14 Test result for test 3.5

Sample of Tests on HP Workstation

We also migrated our simulator to an Itarium 64 bits processor workstation running Linux. We did some tests on 50 qubits on this machine; it took about 18 minutes for 100 runs, while we estimate it would take 360 minutes on a 600MHZ PC. We show a sample of our test results as follows.

String: 11111111110000000000111111111100000000001111111111
Threshold: 30
Error: 1
Same Rotation Error: false

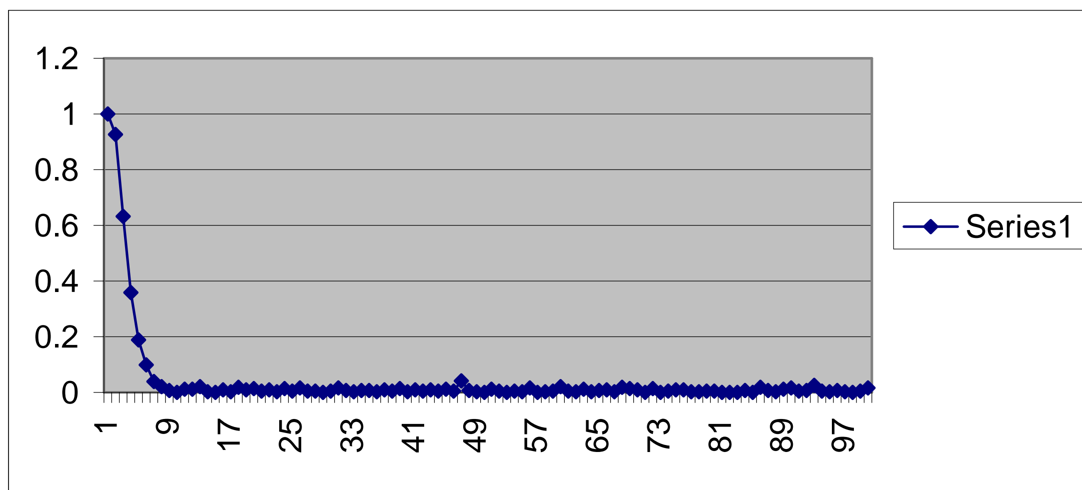


Figure 6-15 Test results on HP workstation

6.3 Test Results Analysis

Many interesting observations can be made from the above test samples. Proving conjectures based on these observations is beyond the scope of the current work. In this section, however, we summarize what we saw.

1. In the case of setting the same error to each rotation operator:

- As Test 2.1 shows, if the threshold equals to the number of bits that is 'on' in the input string, whatever error it is, the output is always '1'. However, as you can see in Test 2.2 and Test 2.3, if the threshold does not equal to the number of bits that is 'on' in the input string, the output is not necessary to be '0'. From these facts, we can conclude that Špalek's algorithm in this case has one-sided error.
- Furthermore, Test 2.2 and 2.3 suggest that although when threshold does not equal to the number of bits that is 'on' the output is not necessarily '0', there are some cycles in just how great the error is. These cycles are related to the length of the input string.
- Given a string length and a Hamming length of this string, we tested all possible strings in Test 3.2, our results show that when threshold does not equal to number of bits that are 'on', all strings have the same output.

2. In the case of different errors are applied to each rotation operator

- Observing Tests 2.4 to 2.7, when we have the threshold value equal to the number of bits that is ‘on’, the value of the error and the length of the input string will both affect the acceptance probability.
- In Test 2.6, when the string length is 8, and the error is no more than 0.05, we have output between 0.96 and 1.0. Increasing the error value causes us to lose the information for acceptance. This is shown in Test 2.4.
- In Test 2.7, when the string length is 30, and the error is no more than 0.013, we have output between 0.96 and 1.0. With increasing the error value, we will lose the information for acceptance, as shown in Test 2.5.
- When the string length does not equal to the number of bits that is ‘on’ in the input string as in Test 2.8 and 2.9, we conclude that as the error gets smaller the output is closer to ‘0’. And, the smaller the length of the input string, the closer the output is to ‘0’.

Chapter 7 Conclusion

In this project, we developed a simulator that simulates a quantum circuit for value gate. The basic algorithm to construct this circuit is Špalek's algorithm, which is described in [HS03]. A naïve implementation of [HS03] takes exponential time and space complexity, and thus is infeasible in the multi-qubits case. We explored an implementation idea that reduces both the time and space complexity to a polynomial quantity. Our simulator uses these ideas, so it runs in reasonable time and space.

In practice, the acceptance model of Špalek's algorithm is unrealistic due to its assumption of exactly prepared quantum states. It is difficult to accurately estimate the effects of errors by calculating from Špalek's algorithm itself. To observe this kind of effect, we introduced an error scheme to our simulator, so our program can simulate more realistic models.

We did tests on inputs up to fifty qubits. Our test results suggest that when applying the same error to each rotation operation, the simulator always produce the correct results whatever the error is. However, when adding different errors there are some ranges for the errors, if the error is out of these ranges then we will lose the ability to give the correct results. This simulator may help further researches on this.

References

[AMP02] F.Ablayev, C.Moore, and C.Pollett. Quantum and Stochastic Branching Programs of Bounded Width. 29th International Colloquium on Automata, Languages, and Programming (ICALP). 2002. p.343--354. ECCC TR02-013.

[FGHP99] S. Fenner, F. Green, S. Homer, and R. Pruim. Determining Acceptance Possibility for a Quantum Computation is Hard for the Polynomial Hierarchy. Proceedings of the Royal Society A (1999) 455, pp 3953--3966.

[GHMP02] F.Green, S.Homer, C.Moore, C.Pollett. Counting, Fanout, and the Complexity of Quantum ACC. Quantum Information and Computation. Vol. 2. No. 1. 2002. pp.35--65.

[HS03] Peter Høyer and Robert Špalek. Quantum Circuits with Unbounded Fan-out. Proceedings of 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2003), Lecture Notes in Computer Science (LNCS 2607), pages 234-246. Springer-Verlag, Berlin, Germany, 2003.

[I94] Circuit Complexity and Neural Networks. Ian Parberry. The Mit Press. 1994.

[NC00] Quantum Computation and Quantum Information. M.Nielson and I.Chuang. Cambridge. 2000.

[Pap94] Computational Complexity. C.Papapdimitriou. Addison Wesley. 1994.

[PS97] P.Shor. Polynomial -time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing 26(5): 1484-1509, 1997.

[Vol00] Introduction to Circuit Complexity. H.Vollmer. Springer-Verlag. 2000.

[Y93] A. C.-C. Yao. Quantum circuit complexity. Proc. 34th IEEE Symposium on Foundations of Computer Science, 352--361, 1993.