Analysis of an Implementation of a Quantum Algorithm for the Triangle Finding Problem

Chen-Fu Chiang*

June 29, 2012

1 Overview

We implement and analyze a quantum algorithm that deployed a discrete quantum walk to find a triangle in a dense graph G. The approach described in the GFI document [1] theoretically follows the work [2] based on the technique designed by Ambainis [3]. Ambainis' approach is to apply a quantum walk on a corresponding Johnson graph based on a given input graph G. However, difficulties arise when implementing such an approach. The main difficulties are (1) generating the uniform superposition to initialize the quantum state, (2) the size of the list of nodes for a single step of quantum walk update is not a power of two and is not contiguous. To overcome the difficulties, the GFI document follows the method [4] proposed by Childs and Kothari by walking on a corresponding Hamming graph based on graph G. As mentioned in the document [1], this implementation avoids using complex data structures but the major issues are the memory, circuit size and timing complexity.

In this report, we describe the formalization of this problem in section 2.1. The structure of the algorithm is explained and illustrated in section 2.2. We provide the estimate of the required logical gates and logical qubits in section 3. The parallization factors and the refined parallization factors are discussed in section 4. We include our source code in Appendix A. The costs of function calls (modules) are individually analyzed in Appendix B and we also include a script that computes the estimate of the resources. The counting of parallelization factors is analyzed and listed in Appendix D along with another script that computes the parallelization factors.

This version of this triangle finding problem, as shown in the GFI document, is one of the detecting/listing as opposed to other versions which count the number of triangles. The guideline document for this implementation is the *GFI Quantum Algorithm Specification for Triangle Finding Problem* with revision number 2011-06-16:10:33:59:ver1.11.

*Département de Physique, Université de Sherbrooke Sherbrooke, Québec, Canada J1K 2R1S Email: Chen-Fu.Chiang@USherbrooke.ca

2 Triangle Finding Problem

2.1 Problem Configuration

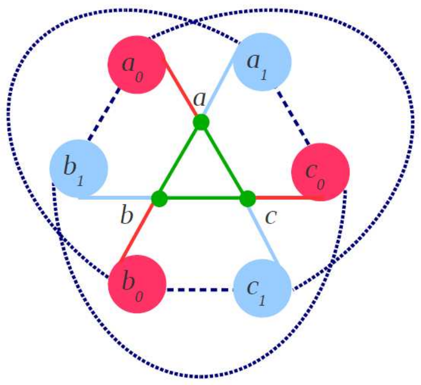

Figure 1: Input Graph G

Given an undirected graph $G(V, E)$ (see figure 1) on N-nodes, we have an oracle function that for any two arbitrary nodes, u and v, tells if (u, v) is an edge in G. In this given problem, we have the nodes divided into 9 subsets. Subsets a, b and c contain only 1 node, respectively. The remaining 6 subsets, a_0, a_1, b_0, b_1, c_0 and c_1, equally contain $\frac{N-3}{6}$ nodes, respectively.

Regarding edges, the subset a (containing only 1 node) of the triangle Δ is connected to each node of the two of the six subsets, one labeled a_0 and the other a_1. Every node in subset a_0 is connected to all the nodes in b_1 and c_1; every node in subset a_1 is connected to all nodes in b_0 and c_0. A similar configuration propagates for subsect b and subset c.

In the configuration, the edges in the given graph G are:

1 $E(a, b) = 1, \quad E(b, c) = 1$ and $E(c, a) = 1$.

2 $E(a, v) = 1, \quad \forall v \in a_0 \cup a_1; \qquad E(b, v) = 1, \quad \forall v \in b_0 \cup b_1; \qquad E(c, v) = 1, \quad \forall v \in c_0 \cup c_1$

2 $E(v, w) = 1 \quad \forall v \in a_0, \forall w \in b_1 \cup c_1; \qquad E(v, w) = 1 \quad \forall v \in b_0, \forall w \in a_1 \cup c_1$

4 $E(v, w) = 1 \quad \forall v \in c_0, \forall w \in a_1 \cup b_1$

The number of edges in Graph G is therefore $|E| = M \approx \frac{N^2}{6}$ and the only triangle in graph G is $\Delta = \{a, b, c\}$. The parameters for this problem are set as $n = 15$[1] and $r = 9$[2][3].

2.2 The Algorithm Structure

The algorithm for quantum walk for the triangle finding problem consists of 3 major parts. The first part is (a) the initialization and the setup for edge information. This includes choosing a starting state $|w\rangle$ and initialising the graph collision quantum walk registers (GCQWRegs). The second part is (b) the core of the algorithm: walking on the corresponding Hamming graph based on the input graph $G(V, E)$. The edge and the nodes information of Graph $G(V, E)$ are encoded in the Oracle function (EdgeORACLE module). The third part of the algorithm is (c) the measurement. It measures two of the registers of the GCQWRegs, which hold the result if the input graph $G(V, E)$ contains at least one triangle.

There are various data structures in this algorithm. They are either arrays of qubits or arrays of integers. The most often used structures are as follows. T is an array of R quNodes. v and w are quNodes of n qubits while i is an index for fetching a particular quNode inside the T array. For instance, in this problem we have 2^{15} nodes in graph G, hence a quNode of 15 qubits is enough to store the identity of any arbitrary node in V. E is a $R(R-1)/2$-qubit register that stores the edge information among the R quNodes in T. GCQWRegs is a collection of 7 different quantum registers (tau, sigma, Ew, iota, cTri, triTestT and triTestTw) that are used in the walks in the Hamming graph. The algorithm consists of the following steps:

[1] $N = 2^n = 2^{15}$ is the number of the nodes in the graph G.

[2] $R = 2^r = 2^9$ is the Quantum Walk parameter that defines the Hamming graph for the walk.

[3] Each quantum walk update in Hamming Graph involves R nodes. They are encapsulated in a list T while list E records the the edge data between those R nodes.

Algorithm 1 Quantum Walk Algorithm for Triangle Finding

Input: The Oracle function contains the information of the input graph G

```
module QWTFP(n,r): (Quantum Walk for Triangle Finding Problem)
1: Initialization reigsters and setup for edge information (for $T$, $E$, $i$, $w$, $GCQWRegs\ GC$)
2: for j := 1, ... $t_m$ do
3:   TestTriangleEdges($T, E, w, n, r, GC$)
4:     TriangleTestT($E, n, r, GC.triTest$)
5:     TriangleEdgeSearch($T, E, w, n, r, 2/3r, GC$)
6:       for k := 1, ... $t_g$ do
7:         GCQWalk($T, E, w, nr, r, 2/3r, GC$)
8:           Set up walk state
9:           for l := 1, ... $tbar_m$ do
10:            Mark $GC.tau$ when conditions are satisfied
11.            for m := 1, ... $tbar_w$ do
12.              GCQWStep
13:     TriangleTestTw($T, E, w, n, r, GC.triTestTw$)
14.   InverseTestTriangleEdges($T, E, w, n, r, GC$)
15.     for o := 1, ... $t_w$ do
16.       QWSH($T, i, v, e, n, r$) (Quantum Walk Step on the Hamming graph)
17. TestTriangleEdges($T, E, w, n, r, GC$)
18. Measure the result
```

Output: Yes/No to tell if the graph contains a triangle

We can visualize the program in the diagram shown in Figure 2.

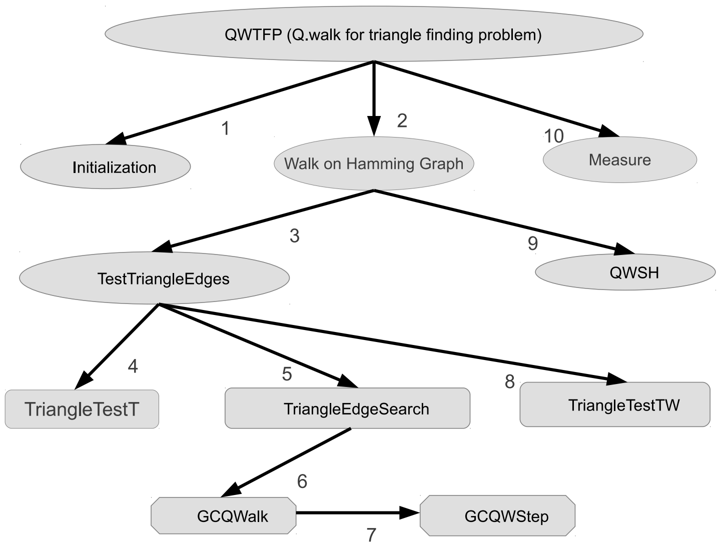

Figure 2: Algorithm Structure

The numbers of iterations are calculated and listed in Table 1. For a more complete table and the relation between those variables, please refer to Appendix B.

variable	n	r	t_m	t_w	$tbar_m$	$tbar_w$	t_g
value	15	9	64	22	8	22	142

Table 1: The Variables and Their Values

3 Resource Estimate

3.1 Assumption

We make the following assumptions:

- An inverse of a unitary U is of the same complexity, in terms of elementary gates, as unitary U.
- Even though the state of a new qubit is $|0\rangle$, we still use the ZERO operation (set qubits to $|0\rangle$) where it appears in the pseudocode.
- The feasibility of implementing a qRAM operation within the quantum circuit model and the physical machine description constraint is not clear [1, 5, 6]. Suppose T is a structure of 2 dimensional array of qubits. To access the qubit located at $T[i][j]$, we must seek sequentially from $T[0]0]$ till $T[i][j]$.

3.2 Required Number of Qubits and Gates

The initial input size of the quantum circuit is contributed by GCQWRegs, T, i, v, E and w. The quantum registers inside GCQWRegs[4] are (1) tau: a $r \times Rbar$-qubit register (2) iota: $rbar$-qubit register (3) sigma : r-qubit register (4) Ew: $Rbar$-qubit register (5) cTri: $(2 \times rbar - 1)$-qubit register (6) triTesT: 1-qubit (7) triTestw: 1-qubit. Hence, the initial input size of the quantum circuit is approximately 145000 qubits[5].

We perform the preliminary resource count for each module (except the main program QWTFP) in Appendix B. We have the following resource estimate for modules summoned by module QWTFP.

Module Name	**CNOT**	**H**	**S**	**T**	**T†**	**X**
RI	16778	7708	0	0	0	4612
SETUP	523264	0	0	0	0	0
TTE	3051293128516	901931321295	450964824828	1352894474484	1803859299312	302180354
QWSH	10257790	3379340	1689646	5068938	6758584	49
FM	0	0	0	0	0	0

Table 2: Resources Estimate for Major Modules in QWTFP. TTE is TestTriangleEdges, FM is final measure and RI is registers initialization.

Based on the algorithm steps described in 1, we know that [6]

$$\text{QWTFP} \approx \text{Registers Initialization} + \text{SETUP} + (2 \times t_m + 1)\ \text{TestTriangleEdges} + (t_m \times t_w)\ \text{QWSH} + \text{Measure}. \qquad (1)$$

[4] $rbar = (2/3)r = 6, Rbar = 2^{rbar} = 64, R = 2^r = 512$

[5] Please refer to QWTFP module in Appendix A for the details of the required quantum registers. In implementation we add one qubit and one r-qubit register for sigma and tau, respectively, for the purpose of performing phase kickback. We also added $R \times r$ qubits for index matching.

[6] Module for getting a minus sign is ignored since it only takes very small resources. Please see Appendix B for details.

Module Name	**ancillae**	**Measure**	**Oracle call**
RI	8389	0	0
SETUP	261632	0	130816
TTE	45133229	0	0
QWSH	559	0	0
FM	0	138511	0

Table 3: Ancillary resources Estimate for Major Modules in QWTFP

We list our computation reults for QWTFP module in Table 4. Notice that the number of ancillae in TestTriangleEdges does not have to be multiplied by the $2 \times t_m + 1$ factor because those ancillae can be reused in next iteration. Our implementation requires a great deal of extra computation space (the ancillae qubits and logical gates). This is due to the fact that the algorithm performs many comparison operations. The comparison operations require a great number of Toffoli gates. This increases the number of required logic gates as a single Toffoli comprises 16 elementary gates. For fudamental modules, such as FetchE, FetchT and StoreT, they all perform index matching sequentially in order to find the right index to perform the fetching (or storing) operations. One of the possible solutions to improve this fundamental issue (the number of ancillae qubits) in the future is to seek the possibility of designing a module that can recycle ancillae qubits by applying the semiclassical approach [7].

Module Name	**CNOT**	**H**	**S**	**T**	**T†**
QWTFP	393631257086926	116353898565483	58176841424380	174530524273140	232707365697520

Table 4: Resources Estimate for QWTFP

Module Name	**X**	**ancillae**	**Measure**	**Oracle call**
QWTFP	38981339270	46190322	138511	14754344

Table 5: Resources Estimate for QWTFP

The ancillae demand comes from the module TriangleTestT, summoned by TestTriangleEdges. TriangleTestT module is the dominant term because it has a 3-level deep loop [7] and each each iteration would require 2 ancillary qubits. We believe that we can greatly reduce the required ancillae by a different implementation (see Appendix C) of algorithm 16 and algorithm 17 with the help of two extra registers of 3 qubits inside module 16 and module 17.

Of course, if the number of available qubits is a major concern, we can perform this task sequentially. In the extreme case, the number of required qubits could be close to the initial input size: 145000 qubits if we do everything sequentially and use extra classical resources to temporarily store the information [7]. This is a matter of tradeoff between the physical resources and time (and decoherence). From a more practical view point, many operations inside this algorithm are for updating the edge array E, which is of size 130816. In terms of time resources and qubit resources, probably 2×145000 (one for the initial input and one for the ancillae) qubits would be the right choice.

[7]$(\sum_{i=0}^{R-1} \sum_{j=i+1}^{R-1} \sum_{k=j+1}^{R-1} 1)$ iterations

Furthermore, with or without the reuse of ancillae, we know theoretically that the query complexity for this algorithm is $O(N^{1.3})$, which is about 741455. If we simply look at the comparison of the Oracle queries, we notice that the empirical data show that the number of our oracle calls is about $19 \times N^{1.3}$, which is quite close to the theoretical result as the factor is only 19.

3.3 Communication Overhead

Based on the logical qubits, we can draw the following graph $\tau(V, E)$. The graph vertices correspond to logical qubits. One new edge connecting the respective vertices is introduced for each two qubit gate (CNOTs). Multiple edges may be introduced for some vertex pairs. Our task is to compute the average degree of the vertices (the same pair of vertices might have several edges), that is $2|E|/|V|$.

We know our total logical qubit is approximately 46190322 and the number of CNOT gates invocations is 393631257086926 (see table 4 and table 5), we have $|E| = 3.9 \times 10^{14}$ and $|V| = 4.6 \times 10^7 + 145000$. Therefore, our average degree of the vertices is $\frac{2\times 3.9\times 10^{14}}{4.6\times 10^7+145000} \approx 1.6 \times 10^7$. The number is so high because we repeatedly reuse the ancillae in the iterations.

4 Parallelization

Although the feasibility of implementing a qRAM operation within the quantum circuit model and the physical machine description constraint is not clear, in this section we investigate the potential of parallelization of the circuit.

4.1 Parallelization Factor

For the parallelization factor, first we assume that operations of only one type can be performed at any given time, but that the same operation can be applied to multiple logical qubits. We then list for each operation the total number of occurrences of the logical gate for the duration of the entire computation, and the average number of logical qubits to which the operation can be applied simultaneously.

In the guiding document, it is assumed that all the qRAM operations are sequential, which limits the possibility of parallelization. If we simply look at the algorithm itself, this algorithm is quite parallelizable (assuming Oracle can take simultaneous queries). We perform the preliminary parallelizable gates count for each module (except QWTFP) in Appendix D. We have the following gate parallelization estimate for modules summoned by module QWTFP. The data is presented in the format of (x,y) where x is the number of different times the gate get invoked and y is the total number of occurance of the corresponding gate. Clearly, for each module its parallel factor is $\frac{y}{x}$ for the corresponding gate.

Module	**CNOT**	**H**	**S**
RI	(2,16778)	(1, 7708)	(0, 0)
SETUP	(4, 523264)	(0, 0)	(0, 0)
TTE	(1635981049840, 3051291585372)	(527147551433, 901932671431)	(263573650472, 450964825112)
QWSH	(568600, 10257730)	(182368, 3379400)	(91182, 1689646)
FM	(0, 0)	(0, 0)	(0, 0)

Table 6: Preliminary Parallelization Observation for Each Module. TTE is TestTriangleEdges, RI is Registers Initialization and FM is Final Measure

Module	T	T†	X	M
RI	(0, 0)	(0, 0)	(2,4612)	(0, 0)
SETUP	(0, 0)	(0, 0)	(0, 0)	(0, 0)
TTE	(790720951416, 1352894475336)	(1054294601888, 1803859300448)	(25856362, 302180354)	(0,0)
QWSH	(273546, 5068938)	(364728, 6758584)	(5, 49)	(0, 0)
FM	(0, 0)	(0, 0)	(0, 0)	(3, 138511)

Table 7: Preliminary Parallelization Observation for Each Module

Based on Eqn. (1) and table 6, we can compute the parallelization factors [8]. It is no surprise that many gates (such as the S gate, the T gate and the $T^\dagger$ gate) have the same parallelization factor because they are summoned when Toffoli gate is summoned. Hence, we can further infer that Toffoli gate has the same paraellization factor. As for the H gate and the CNOT gate, they are summoned mainly, in terms of number of invocations, in the TestTriangleEdges module. The TestTriangleEdges module is dominated by the TriangleEdgeSearch moudle where it invokes a great number of Toffoli gates. Therefore, their parallel factor is close to the parallel factor of Toffoli gates. As mentioned earlier, this algorithm could be parallelizable provided that a qubit (and the Oracle) can be accessed (read) simultaneously and many extra ancillary qubits (without reusing the ancillary qubits).

The parallelization could come from the TriangleTestT and the TriangleTestTw modules. They both have large number of iterations and those iterations could be in parallel. The factor for X gate is high as it is used mostly in the EvaluateOR module while TriangleTestT and the TriangleTestTw modules invoke EvaluateOR module frequently.

Logical Operation	Occurrance	Parallelization Factor
H	1.16×10^{14}	1.71
S	$5.81 \times \times 10^{13}$	1.71
T	1.74×10^{14}	1.71
T†	2.32×10^{14}	1.71
X	3.89×10^{10}	11.68
CNOT	3.93×10^{14}	1.86
Measure	1.38×10^{5}	46170

Table 8: Parallelization Factor

4.2 Refined Parallelization Factor

Regarding the refined parallelization factor, for each epoch we state the types of logical gates that can be used in that epoch, the number of repetition of that epoch (in each repetition any logical gate of an allowed type can be applied to any of the logical qubits), and for each allowed gate the relative frequency of its occurrence in the epoch.

In chronological order, the algorithm has three parts, initialization (including SETUP), walking on the Hamming graph and measurement. We will divide the algorithm into four epochs. Since we know

$$\text{QWTFP} \approx \underbrace{Registers\quad Initialization}_{\text{epoch 1}} + \underbrace{SETUP}_{\text{epoch 2}} + \underbrace{(2 \times t_m + 1) TestTriangleEdges + (t_m \times t_w) QWSH}_{\text{epoch 3}} + \underbrace{Measure}_{\text{epoch 4}},$$

[8]See the script file attached in Appendix D for the computation details

we can simply look up the values in table 2 and table 4 for each epoch. After some simple computations, we list the results of the refined parallelization factors in table 9[9].

Epoch Number	**Repetitions**	**Allowed Frequency**
1	2.91×10^4	CNOT (57.7%), H (26.5%), X (15.8%)
2	5.23×10^5	CNOT (100%)
3	9.75×10^{14}	CNOT (40.354%), H (11.928%), S (5.964%), T (17.893%) $T^\dagger$ (23.857%), X (0.004%)
4	1.38×10^5	Measure (100%)

Table 9: Refined Parallelization Factor

References

[1] R. Wisniewski, *Quantum Algorithm Specification: Triangle Finding Problem*, GFI document: Revision 2011-06-16:10:33:59:ver1.11

[2] F. Magniez, M. Santha and M. Szegedy, *Quantum Algorithms for the Triangle Problem*, quant-ph/0310134 version 2

[3] A. Ambainis, *Quantum Walk Algorithm for Element Distinctness*, quant-ph/0311001v8

[4] A. Child and R. Kothari, *Quantum Query Complexity of Minor-closed Graph Properties*, quant-ph/arXiv:1011.1443.v1

[5] V. Giovannetti, S. Lloyd and L. Maccone, *Quantum Random Access Memory*, quant-ph/0708.1879v2

[6] V. Giovannetti, S. Lloyd and L. Maccone, *Achitecture for a Quantum Random Access Memory*, quant-ph/0807.4994v2

[7] R. Griffiths and C. Niu, *Semiclassical Fourier Transform for Quantum Computation*, Phys. Rev. Lett. 76, 3228 (1996).

[9] Take CNOT in epoch 3 for instance, we compute the total CNOT gates used in epoch 3 as $\sum_{CNOT}$. Similar computation for Measure, H, Z, X. Let $\sum_{all} = \sum_{CNOT} + \sum_{H} + \sum_{Z} + \sum_{X} + \sum_{Measure}$. The percentage of CNOT gates is thus obtained via $\sum_{CNOT} / \sum_{all}$.

A Source Code

In this section, we include our source code (for Scaffold language) for this algorithm. The code consists of both the algorithm for the triangle finding problem and the algorithm for the Oracle function. The code is based on the pseudocode given in the GFI document [1].

```
/***************************************************************************************
* File: TriangleFindingProblem.quantum
*
* Author: Chen-Fu Chiang @ USherbrooke
*
* Implementation of the triangle finding algorithm specified in the
* Quantum Computer Science Program @ Government Furnished Information document
*
* Document date: June 20, 2011
* Document Version: 1.0.0
* Document Reversion number: 188
*
* Implementation date: 2012
****************************************************************************************/
// Commonly used gates
gate H(qreg q[1]);
gate X(qreg q[1]);
gate Z(qreg q[1]);
gate S(qreg q[1]);
gate T(qreg q[1]);
gate Tdag(qreg q[1]);
gate Sdag(qreg q[1]);
gate CNOT(qreg target[1], qreg control[1]);



// Global variables
/**************************************************************
#define n  15
#define N  32768        // N = 2 ^ n
#define r  9            // r = (3/5)n
#define R  512          // R = 2 ^ r
#define RR1 130816      // RR1 = R(R-1)/2
#define rbar  6         // rbar = (2/5)n
#define Rbar  64        // Rbar = 2^rbar
#define CTR  11         // CRT = 2*rbar - 1
#define tm 64           // Quantum walk iterations (floor of N/R)
#define tw 22           // Quantum walk step iterations (floor of sqrt(R))
#define tg  142         // Grover iteration (pi/4)* (Sqrt(N))
#define tbarm 8         // R/Rbar
#define tbarw 22        // (floor of sqrt(R))
***************************************************************/


/**************************************************************
*       Useful Modules
***************************************************************/
module Swap(qreg q1[1],qreg q2[1])
{
        CNOT(q2[0],q1[0]);
        CNOT(q1[0],q2[0]);
        CNOT(q2[0],q1[0]);
}

module ctrSwap(qreg q1[1],qreg q2[1], qreg q3[1])
{
        Toffoli(q2[0],q1[0], q3[0]);
        Toffoli(q1[0],q2[0], q3[0]);
        Toffoli(q2[0],q1[0], q3[0]);
}

module CR(qreg target[1], qreg control[1], double theta[1])
{
        R(target[0], theta / 2.0);
        CNOT(target[0], control[0]);
        R(target[0], -1.0 * theta / 2.0);
        CNOT(target[0], control[0]);
}

module Toffoli(qreg target[1], qreg ctr2[1], qreg ctr1[1])
{
        H(target[0]);
        CNOT(target[0],ctr2[0]);
        Tdag(target[0]);
        CNOT(target[0], ctr1[0]);
        T(target[0]);
        CNOT(target[0], ctr2[0]);
        Tdag(target[0]);
        CNOT(target[0], ctr1[0]);
        Tdag(ctr2[0]);
        T(target[0]);
        CNOT(ctr2[0], ctr1[0]);
        H(target[0]);
        Tdag(ctr2[0]);
        CNOT(ctr2[0], ctr1[0]);
        T(ctr1[0]);
        S(ctr2[0]);
}

/**********************************************************
*       target[0] = OR(ctr2[0] , ctr1[0])
**********************************************************/
module EvaluateOR(qreg target[1], qreg ctr2[1], qreg ctr1[1])
{
        X(ctr1[0]);
        X(ctr2[0]);
        Toffoli(target[0], ctr2[0], ctr1[0]);
        X(target[0]);
}

module InvEvaluateOR(qreg target[1], qreg ctr2[1], qreg ctr1[1])
{
        X(target[0]);
        Toffoli(target[0], ctr2[0], ctr1[0]);
        X(ctr2[0]);
        X(ctr1[0]);
}

/*****************************************************************
* Determine if a Node is a |00...0> and store the result in qubit
* Inverse of it must be run (after getting the final evaluation)
* to transform the input bits back
*****************************************************************/
module ZeroNodeTest(quNode w, qreg dum[n-1], int n)
{
        int j = 0;
        for(j = 0; j< n; j++)
        {
                X(w.reg[j]);
        }

        Toffoli(dum[0], w.reg[1], w.reg[0]);
        for(j = 1; j < n-2; j++)
        {
                Toffoli(dum[j], dum[j-1], w.reg[j+1]);
        }

        Toffoli(dum[n-2], dum[n-3], w.reg[n-1]);
}

module InvZeroNodeTest(quNode w, qreg dum[n-1], int n)
{
        int j = 0;
        Toffoli(dum[n-2], dum[n-3], w.reg[n-1]);

        for(j = n-3; j > -1; j--)
        {
                Toffoli(dum[j], dum[j-1], w.reg[j+1]);
        }

        Toffoli(dum[0], w.reg[1], w.reg[1]);

        for(j = 0; j< n; j++)
        {
                X(w.reg[j]);
        }
}

/****************************************************************************
* Here we reuse the ancillae ctrbit[14] to see if input input[k] is all 0s Reg
****************************************************************************/
module ZeroRegTest(qreg input[k], qreg dum[n-1], int k)
{
        int j = 0;
        for(j = 0; j< k; j++)
        {
                X(input[j]);
        }

        Toffoli(dum[0], input[1], input[0]);
        for(j = 1; j < k-2; j++)
        {
                Toffoli(dum[j], dum[j-1], input[j+1]);
        }

        Toffoli(dum[k-2], dum[k-3], input[k-1]);
}


module InvZeroRegTest(qreg input[k], qreg dum[n-1], int k)
{
        int j = 0;
        Toffoli(dum[k-2], dum[k-3], input[k-1]);

        for(j = k-3; j > -1; j--)
        {
                Toffoli(dum[j], dum[j-1], input[j+1]);
        }

        Toffoli(dum[0], input[1], input[0]);

        for(j = 0; j< k; j++)
        {
                X(input[j]);
        }
}

/*****************************************************************************
```

```
*  Here we reuse the ancillae ctrbit[14] to see if input is an all 1s Register
******************************************************************************/
module OneRegTest(qreg input[k], qreg dum[n-1], qreg ctrNode[1], int k)
{
        int j = 0;

        Toffoli(dum[0], input[1], input[0]);
        for(j = 1; j < k-2; j++)
        {
                Toffoli(dum[j], dum[j-1], input[j+1]);
        }

        Toffoli(dum[k-2], dum[k-3], input[k-1]);

        CNOT(qreg ctrNode[1], dum[k-2]);
}

module InvOneRegTest(qreg input[k], qreg dum[n-1], qreg ctrNode[1], int k)
{
        int j = 0;

        CNOT(qreg ctrNode[1], dum[k-2]);

        Toffoli(dum[k-2], dum[k-3], input[k-1]);

        for(j = k-3; j > -1; j--)
        {
                Toffoli(dum[j], dum[j-1], input[j+1]);
        }

        Toffoli(dum[0], input[1], input[0]);
}
/************************************************
Evaluate the qubit value in terms of boolean
************************************************/
boolean module evaluateToBool(qreg reg[1])
{
        int ret = 0;
        qreg dreg[1];

        CNOT(dreg[0], reg[0]);
        measX(dreg[0], ret);

        if(ret == 1)
                return true;

        else
                return false;
}

// Replaced by CompareIJ
/************************************************
* Convert a r-bit qreg into integer presentation
************************************************/
int module quIntrToInt(qreg ireg[r], greq treg[r], dum[n-1])
{
        int ret = 0;
        int i = 0;
        int j = 0;

        for(i = 0; i < r ; i++)
        {
                if(evaluateToBool(reg[i]))
                        j = 1;
                else
```

```
                        j = 0;

                ret += j*pow(2,i);
        }

        return ret;
}

//Replaced by CompareIJrbar
/*********************************************************
* Convert a 6-bit (rbar-bit) qreg into integer presentation
*********************************************************/
int module quInt6ToInt(qreg reg[6])
{
        int ret = 0;
        int i = 0;
        int j = 0;

        for(i = 0; i < 6 ; i++)
        {
                if(evaluateToBool(reg[i]))
                        j = 1;
                else
                        j = 0;

                ret += j*pow(2,i);
        }

        return ret;
}

/*********************************************************
* CompareIJ
* Para: quantum integer ireg, array index idxIJ (note here
* the idxIJ is an element of type quIntr, not an array of quIntr)
* ctrbit2
* Calls CompareIndexr to get the bitwise comparision of indices
* Calls OneRegTest to see if the bitwise comparison is all 1
* Because of this module is called by many elementary comparison
* operations, such as FetchT, FetchE, and etc. Hence,
* we need to pass those parameters (idxIJ, ctrbit2, ctrbit, ctrNode)
* to many modules, even these 4 parameters are not specified in the
* GFI document
*********************************************************/
module CompareIJ(qreg ireg[r], quIntr idxIJ, qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        CompareIndexr(ireg, idxIJ, ctrbit2);
        OneRegTest(ctrbit2, ctrbit, ctrNode, r);
}

module InvCompareIJ(qreg ireg[r], quIntr idxIJ, qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        InvOneRegTest(ctrbit2, ctrbit, ctrNode, r);
        InvCompareIndexr(ireg, idxIJ, ctrbit2);
}

/*********************************************************
* CompareIndexr
* See if two indices of size r are the same
* Para: ireg and treg are the two indices we are comparing
*               dum is the global ctrbit2[14] we can reuse
*               each bit comparison is written to dum array
*********************************************************/
module CompareIndexr(qreg ireg[r], quIntr treg, qreg dum[n-1])
{
        int i = 0;

        for (i = 0; i < r; i++)
        {
                CNOT(treg.reg[i], ireg[i]);
                CNOT(dum[i], treg.reg[i]);
        }
}

/**********************************************************
* InvCompareIndexr
* Inverse of CompareIndexr
* Para: ireg and treg are the two indices we are comparing
*               dum is the global ctrbit2[14] we can reuse and it
*               records the bit-wise comparison
**********************************************************/
module InvCompareIndexr(qreg ireg[r], quIntr treg, qreg dum[n-1])
{

        int i = 0;

        for (i = r-1; i > -1; i--)
        {
                CNOT(dum[i], treg.reg[i]);
                CNOT(treg.reg[i], ireg[i]);
        }
}

/**********************************************************
* CompareIJrbar
* Similar to CompareIJ but the index is now
* (note here the idxIJ is an element of type quIntr, not an array of quIntr)
* of length rbar
**********************************************************/
module CompareIJrbar(qreg ireg[rbar], quIntr idxIJ, qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        CompareIndexrbar(ireg, idxIJ, ctrbit2);
        OneRegTest(ctrbit2, ctrbit, ctrNode, rbar);
}

module InvCompareIJrbar(qreg ireg[rbar], quIntr idxIJ, qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        InvOneRegTest(ctrbit2, ctrbit, ctrNode, rbar);
        InvCompareIndexrbar(ireg, idxIJ, ctrbit2);
}

/**********************************************************
* CompareIndexrbar
* Similar to CompareIndexr but the length is rbar
**********************************************************/
module CompareIndexrbar(qreg ireg[rbar], quIntr treg, qreg dum[n-1])
{
        int i = 0;

        for (i = 0; i < rbar; i++)
        {
                CNOT(treg.reg[i], ireg[i]);
                CNOT(dum[i], treg.reg[i]);
        }
}

/**********************************************************
* InvCompareIndexrbar
* Similar to InvCompareIndexrbar but the length is rbar
**********************************************************/
```

```
module InvCompareIndexrbar(qreg ireg[rbar], quIntr treg, qreg dum[n-1])
{

        int i = 0;

        for (i = rbar-1; i > -1 ; i--)
        {
                CNOT(dum[i], treg.reg[i]);
                CNOT(treg.reg[i], ireg[i]);
        }
}


/**********************************************************
* ctrCTRincr
* Increment the GC.cTri by 1; considering element at
* location 0 is the least significant bit. So we are
* starting from the most significant bit (use of toffoli being
* controlled by less significant bits), repeat this process
* till we do CNOT on the 2nd least significant bit (controlled by
* the least significant bit) and then finally do a CNOT operation
* on the least significant bit controlled by the ctrbit. The reverse of
* this operation would be a decrement.
* Assuming that overflow case will not occur
**********************************************************/
module ctrCTRincr(qreg cTri[2*rbar-1], qreg ctrQ[1], qreg ctrbit2[n-1], qreg ctrNode[1])
{
        int i = 0;
        int j = 0;

        /**************************************************************************
        * we do 2*rbar -1 but actually we only need to run to 2*rbar-2
        * the value of if the first t elements (0 till t-1) in cTri are all 1s is stored at
        * ctrbit2[t-2] where t should be greater than 1
        **************************************************************************/

        OneRegTest(cTri, ctrbit2, ctrNode[0], 2*rbar-1);
        for(i = 2*rbar - 2; i > 1; i--)
        {
                CNOT(cTri[i], ctrbit2[i-2]);
        }

        ZERO(ctrNode[0]);
        for(j = 0; j< n-1; j++)
        {
                ZERO(ctrbit2[j];
        }
        CNOT(cTri[1], cTri[0]);
        CNOT(cTri[0], ctrQ[0]);
}

/**********************************************************
* ctrCTRdecr
* decrement the GC.cTri by 1; similar to increment case but
* we have to test the all 0s case, instead of all 1s case.
* Assuming that overflow case will not occur
**********************************************************/
module ctrCTRdecr(qreg cTri[2*rbar-1], qreg ctrQ[1], qreg ctrbit2[n-1], qreg ctrNode[1])
{
        int i = 0;
        int j = 0;

        ZeroRegTest(cTri, ctrbit2, ctrNode[0], 2*rbar-1);

        for(i = 2; i < 2*rbar-1; i++)
        {
                /************************************************************************
                * have to do so because in the Zero test we perform X operation
                * and since we are not going to do the inverse of that test, we need to
                * make sure elements in cTri are restored to it original values
                * we do not restore element at 0 and 1 now because we will do that later
                ************************************************************************/
                X(cTri[i]);
        }

        for(i = 2*rbar - 2; i > 1; i--)
        {
                CNOT(cTri[i], ctrbit2[i-2]);
        }

        ZERO(ctrNode[0]);
        for(j = 0; j< n-1; j++)
        {
                ZERO(ctrbit2[j];
        }

        // restoring element at location 1
        X(cTri[1]);
        CNOT(cTri[1], cTri[0]);

        // restoring element at location 0
        X(cTri[0]);
        CNOT(cTri[0], ctrQ[0]);
}

/************************************************
* Measure a node
************************************************/
module nodeMeasure(quNode w, int ret[n])
{
        int i;

        for(i = 0; i < n ; i++)
        {
                measX(w.reg[i], ret[i]);
        }
}


/************************************************
* Measure array of nodes and store the value
************************************************/
module TMeasure(quNode T[R], int res[R][n])
{
        int i;
        int j;

        for(i = 0; i < R ; i++)
        {
                for(j = 0; j< n; j++)
                {
                        measX(T[i].reg[j], res[i][j]);
                }
        }
}


/******************************************************
* Measure array of edge connection and store the value
******************************************************/
```

```
module EMeasure(qreg E[RR1], int fin[RR1])
{
        int i;

        for(i = 0; i < RR1 ; i++)
        {
                        measX(E[i], fin[i]);
        }
}


/************************************************************
*       Data Structures
************************************************************/
struct quNode
{ qreg reg[n];
  qreg phase[1];
};

struct GCQWRegs
{
        // extra 1 bit (or qIntr) is needed for recording the phases
        qIntr tau[Rbar+1];
        qreg  sigma[r+1];
        qreg  iota[rbar];
        qreg  Ew[Rbar];
        qreg  cTri[2*rbar-1];
        //qubit holding test results for the triangle in T
        qreg  triTestT[1];
        //qubit holding test results for the triangle in T and w
        qreg  triTestw[1];
};

struct quInt
{ qreg reg[31];
};

struct quIntr
{ qreg reg[r];
};


/******************************************************************
*       Algorithm 1: QWTFP
*       Para: n: 2^n is the number of nodes in the graph
*             r: 2^r is the size of R-Tuples in the Hamming graph H(V,R)
*       Return:  Yes/No if the graph has a triangle
******************************************************************/
int module QWTFP(int n, int r)
{
        int j = 0;
        int k = 0;
        int l = 0;

        // Required quantum registers (the initial circuit input)
        // T: 512 * 16; v:15; E:130816; w:16 (1 extra bit for phase);
        // testTEdge: 1; test: 1; GC: 678
        // Hence, the total initial quantum circuit input is approx 145000

        quNode T[R];
        qreg i[r];
        quNode v;
        qreq E[RR1];
        quNode w;
        qreg testTEdge[1];
        qreg test[1];
        qreg TEPhase[1];
        qreg ctrbit[14];
        qreg ctrbit2[14];
        qreg ctrNode[1];

        /*********************************************************************
        * used for index matching
        * each element inside idxIJ (let say idxIJ[k]) is physically wired
        * close to the quNode T[k] such that when and index i is given,
        * we compare i and idxIJ[k] for k = 0 ... 511 until we find a match
        * Once match is found, we continue with the corresponding T[k] to
        * proceed with corresonding operation.
        * The initialization of this index, in physical implementation, is not
        * needed because we can just hardwire the NOT gates (to turn 0 to 1)
        * to let idxIJ[k] to represent the number k (in binary presentation).
        * However, in simulation, we need to do the initialization.
        *********************************************************************/
        quIntr idxIJ[R];

        // used for initializing idxIJ from 0 to 511
        int bin[r];

        // the quotient
        int quo = 0;

        // Graph Collision Register
        GCQWRegs GC;

        int testTMeasure = 0;

        int ret[n];
        int res[R][n];
        int fin[RR1];

        ZERO(test);
        ZERO(testTEdge);


        for( j = 0; j < R; j++)
        {
                for(l = 0; l < r; l++)
                        bin[l] = 0;


                for(k = 0; k < n; k++)
                {
                        INITIALIZE(T[j].reg[k]);
                }


                // get the binary presentation of the number j
                quo = j;

                for(l = 0; l < r; l++)
                {
                        bin[l] = quo%2;
                        quo = quo/2;
                        if(quo==0)
                        {break;}
                }

                /*************************************************************************
                * Initialize idxIJ[j]; this loop could have been merged with previous loop
                *  used for the binary presentation but for readability, we separate them.
                ****************************************************************************/
                for(l = 0; l < r; l++)
                {
                        if(bin[l] == 1)
                                X((idxIJ[j]).reg[l]);
                }
        }

        for(j=0; j<r; j++)
        {
                INITIALIZE(i.reg[j]);
        }

        for(j = 0; j<n; j++)
        {
                INITIALIZE(v.reg[j]);
        }

        // To generate the (|0>-|1>)/sqrt(2) state for recording phase later
        H(X(v.phase[0]));

        for(j = 0; j < n; j++)
        {
                ZERO(w.reg[j]);
        }

        // To generate the (|0>-|1>)/sqrt(2) state for recording phase later
        H(X(w.phase[0]));

        for(j = 0; j < Rbar; j++)
        {
                for(k = 0; k < r; k++)
                {
                        ZERO((GC.tau[j]).reg[k]);
                }
        }

        // To generate the (|0>-|1>)/sqrt(2) state for recording phase later
        // The phase is encoded at the last (extra qIntr) element's (r qubits) first qubit
        H(X((GC.tau[Rbar]).reg[0]));

        for(j = 0; j < rbar; j++)
        { ZERO(GC.iota[j]);}

        for(j = 0; j < r; j++)
        { ZERO(GC.sigma[j]);}

        // To generate the (|0>-|1>)/sqrt(2) state for recording phase later
        H(X(GC.sigma[r]));


        for(j = 0; j< Rbar; j++)
        {ZERO(GC.Ew[j]);}

        for(j = 0; j < 2*rbar-1; j++)
                {ZERO(GC.cTri[j]);}

        ZERO(GC.triTestT);
        ZERO(GC.triTestTw);

        // Set up E with the edget information for T
        SETUP(E,T,R);

        // Quantum Walk on the Hamming Graph
        for(j = 0; j < tm; j++)
```

```
        {
                // Test T for triangle edges
                TestTriangleEdges(T,E,w, n, r, GC, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                EvaluateOR(ctrbit[0], GC.triTestT, GC.triTestTw);

                // Phase flip for TE
                CNOT(TEPhase[0], ctrbit[0]);
                InvEvaluateOR(ctrbit[0], GC.triTestT, GC.triTestTw);

                //TODO: Assume the inverse function will be present when provided the original circuit
                InvTestTriangleEdges(T,E,w,n,r,GC,  idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                // Quantum Walk on H(V,R)
                for(k = 0; k < tw; k++)
                {
                        QWSH(T,i, v, E, n, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
                }
        }


        // Final Check for triangle edges in T
        TestTriangleEdges(T,E,w,n,r,GC);


        // if GC.triTestTw is 1, then testTMeasure should be set to 1
        EvaluateOR(ctrbit[0], GC.triTestT, GC.triTestTw);
        ctrONE(testTEdge[0], ctrbit[0]);
        InvEvaluateOR(ctrbit[0], GC.triTestT, GC.triTestTw);
        measX(testTEdge, testTMeasure);

        // Get the triangle information if the testTMeasure is 1 (there is a triangle)
        if(testTMeasure==1)
        {
                //meansure T, E and w
                nodeMeasure(w, ret);
                TMeasure(T, res);
                EMeasure(E, fin);
        }

        // return YES/NO ; if yes, then we need to check ret, res and fin
        return testTMeasure;
}


/*************************************************************
*       Algorithm 2-0: ONE
*       Initializes the qubits in a register to 1
*       Para: qreg reg[] of size 1
*************************************************************/
module ONE(qreg reg[1])
{
        qreg dreg[1];
        CNOT(dreg[0], reg[0]);
        X(dreg[0]);
        CNOT(reg[0], dreg[0]);
}

/*************************************************************
*       Algorithm 2-1: ctrONE
*       Initializes the qubits in a register to 1
*       Para: qreg reg[] of size 1
*************************************************************/
module ctrONE(qreg reg[1], qreg test[1])
{
        qreg dreg[1];
        Toffoli(dreg[0], reg[0], test[0]);
        CNOT(dreg[0], test[0]);
        Toffoli(reg[0], dreg[0], test[0]);
}


/***************************************************************
*       Algorithm 2: ZERO
*       Initializes the qubits in a register to 0
*       Para: qreg reg[] of size n
***************************************************************/
module ZERO(qreg reg[1])
{
        qreg dreg[1];
        CNOT(dreg[0], reg[0]);
        CNOT(reg[0], dreg[0]);
}


/************************************************************************
*       Algorithm 3: INITIALIZE
*       Sets each qubit in the n-qubit register reg to 0 and
*       then calls the Hadamard gate to place reg into superpostion
*       Para: qreg reg[] of size 1
*       Comment: Along with a loop, we can use this module for
*       to perform INITIALIZE work on input register of size greater than 1.
************************************************************************/

 module INITIALIZE(qreg reg[1])
{
        ZERO(reg[0]);
        H(reg[0]);
}


/***************************************************************
*       Algorithm 4: HADAMARD
*       Applies the Hadamard primitive to each qubit in the register
*       Para: qreg reg[] of size n
*       Comment: This modules applies the Hadamard primitive to each
*       qubit in the register, that is a combination of for loop
*       of size n and n uses of Hadamard primitive. Whenever we
*       encounter the invocation of this module, we directly
*       use a combination of for loop along with the Hadamard
*       primitive (H) to perform such a task.
***************************************************************/



/***************************************************************
*       Algorithm 5: SETUP
*       Para: E a (R(R-1)/2)-qubit reg
*         T a quNode[R]register:
*             T[j] a quNote for j = 0,1,..., R-1
***************************************************************/
module SETUP(qreg E[RR1], quNode T[R], int R)
{
        int j = 0;
        int k = 0;
        int jk = 0;
        int l = 0;

        for(l = 0; l < RR1; l++)
        {
                ZERO(E[l]);
        }


        //Comment: This step seems redundant since previous step already ZERO all elements
        for(j = 0; j < R-1; j++)
        {
                for(k=j+1; k < R; k++)
                {
                        jk = ((k*(k-1))/2) + j;
                        ZERO(E[jk]);

                        // Note: In Oracle implementation, the oracle takes 3 parameters;
                        // hence, we have 3 parameters here.
                        EdgeORACLE(T[j], T[k], E[jk]);
                }
        }
}


/***************************************************************
*       Algorithm 6
*       Para: T an array of quNodes
*             i a quInt register
*             v a quNode register (that is n qubits)
*             E a (R(R-1)/2)-qubit register
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
***************************************************************/

module QWSH(quNode T[R], qreg i[r], quNode v, qreg E[RR1], int n, int r,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        quNode Td;
        qreg Ed[R];
        int i;

        for(i = 0; i<n; i++)
        {
                ZERO(Td[i]);
                ZERO(Ed[i]);
        }

        //Diffuse on i and v registers
        DIFFUSEVI(v, i, n+r);

        FetchT(i, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        FetchStoreE(i, E,Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        UPDATE(T, Td, Ed, r);

        StoreT(i, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        SWAP(Td, v, n);

        StoreT(i, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        UPDATE(T, Td, Ed, r);

        // Note: the GFI has an extra R parameter that is a mistake as
        // FetchStoreE only takes 4 parameters and R = 2^r
        FetchStoreE(i, E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
        // Note: the GFI has an extra R parameter that is a mistake
        // as FetchT only takes 5 parameters and R = 2^r
        FetchT(i, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
}


/**************************************************************
*       Algorithm 7
*       Para: w a quNode
*             n an given number [size of w]
**************************************************************/
module DIFFUSENode(quNode w, int n)
{
        int j = 0;
        for(j = 0; j < n; j++)
        {
                H(w.reg[j]);
        }
                // get a minus sign for the state
                X(w.phase[0]);

                //now need to reset the minus sign if the stat is |000...0>
                ZeroNodeTest(w, ctrbit, n);
                CNOT(w.phase[0], ctrbit[n-2]);
                InvZeroNodeTest(w, ctrbit, n);

        for(j = 0; j < n; j++)
        {
                H(w.reg[j]);
        }
}


/**************************************************************
*       Algorithm 7-1
*       Para: alpha: quNode of size n
*                beta: quantum registers of size r
*                toal: sum of n and r
**************************************************************/
// perform Diffuse on two registers of size n (thi is quNode) and r, respectively
module DIFFUSEVI(quNode alpha, qreg beta[r], int tol)
{
        int j = 0;
        int i = 0;

        // since quNode is always of size n
        for(j = 0; j < n; j++)
        {
                H(alpha.reg[j]);
        }

        for(i = 0; i < r; i++)
        {
                        H(beta[i]);
        }

        X(alpha.phase[0]);


        ZeroNodeTest(alpha, ctrbit, n);
        // ctrbit2 is also of size n-1, hence, we have extra n-r = 6 bits left after ZeroRegTest
        ZeroRegTest(beta, ctrbit2, r);
        Toffoli(ctrbit2[r-1], ctrbit2[r-2], ctrbit[n-2]);
        CNOT(alpha.phase[0], ctrbit2[r-1]);
        //uncompute
        Toffoli(ctrbit2[r-1], ctrbit2[r-2], ctrbit[n-2]);
        InvZeroRegTest(beta, ctrbit2, r);
        InvZeroNodeTest(alpha, ctrbit, n);


        for(j = 0; j < n; j++)
        {
                H(alpha[j]);
        }

        for(i = 0; i < r; i++)
        {
                H(beta[i]);
        }
}


/*********************************************************************
*       Algorithm 7-2
*       Para: ta, ma:  are quantum registers(iota and sigma) of GCQWRegs
*                 tol: sum of sizes of iota and sigma
*********************************************************************/

// perform Diffuse on two registers of size rbar and r, respectively
module DIFFUSEGC(qreg ta[rbar], qreg ma[r+1], int tol)
{
        int i  = 0;
        int j = 0;
        for(i = 0; i < rbar; i++)
        {
                H(ta[i]);
        }

        for(j = 0; j < r; j++)
        {
                H(ma[j]);
        }

        // get minus sign for all states
        X(GC.sigma[r]);

        // remove minus sign for |00..0>
        ZeroRegTest(ta, ctrbit, rbar);
        // ctrbit2 is also of size n-1, hence, we have extra n-r = 6 bits left after ZeroRegTest
        ZeroRegTest(ma, ctrbit2, r);
        Toffoli(ctrbit2[r-1], ctrbit2[r-2], ctrbit[rbar-2]);
        CNOT(GC.sigma[r], ctrbit2[r-1]);

        //uncompute
        Toffoli(ctrbit2[r-1], ctrbit2[r-2], ctrbit[rbar-2]);
        InvZeroRegTest(ma, ctrbit2, r);
        InvZeroRegTest(ta, ctrbit, rbar);

        for(i = 0; i < rbar; i++)
        {
                H(ta[i]);
        }

        for(j = 0; j < r; j++)
        {
                H(ma[j]);
        }
}
```

```
/*************************************************************
*       Algorithm 8
*       Para: I quIntr
*             T an array of R quNodes
*             Td a quNode register (that is n qubits)
*             r integer (given)
*         n integer (given)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*************************************************************/
module FetchT(qreg I[r], quNode T[R], quNode Td, int r, int n, quIntr idxIJ[R],
        qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;

        // depreciated
        // int i = quIntrToInt(I);

        for(j = 0; j < R; j++)
        {
                CompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);

                    for(k = 0; k < n; k++)
                        {
                                Toffoli(Td.reg[k], T[j].reg[k], ctrNode[0]);
                        }
                InvCompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);

        }
}

/*************************************************************
*       Algorithm 8-1
*       Para: I quInt
*             T an array of R qubits
*             Td an array of Rbar qubits
*             r integer (given)
*         b integer (in this case it is 1)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*************************************************************/

module FetchTr1(qreg I[r], qreg T[R], qreg Td[1], int r, int b, quIntr idxIJ[R],
        qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;

        // depreciated
        // int i = quIntrToInt(I);

        for(j = 0; j < R; j++)
        {
                CompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
                for(k = 0; k < b; k++)
                        {
                                // in this case, k is always 0
                                Toffoli(Td[k], T[j], ctrNode[0]);
                        }
                InvCompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);

        }
}


/**************************************************************
*       Algorithm 8-2
*       Para: iota      quInt
*             tau an array of R qubits
*             Td an array of Rbar qubits
*             rbar integer (given)
*         r integer (given)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
**************************************************************/
module FetchTrbarr(qreg iota[rbar], quIntr tau[Rbar], qreg Td[r], int rbar, int r,
        quIntr idxIJ[R], qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;
        // depreciated
        // int i = quInt6ToInt(iota);

        for(j = 0; j < Rbar; j++)
        {
                CompareIJrbar(iota, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
                        for(k = 0; k < r; k++)
                        {
                                // in this case, k is always 0
                                Toffoli(Td[k], tau[j].reg[k], ctrNode[0]);
                        }
                InvCompareIJrbar(iota, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
        }
}



/**************************************************************
*       Algorithm 9
*       Para: i quInt
*             T an array of R quNodes
*             Td a quNode register (that is n qubits)
*             r integer (given)
*             n integer (given)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
**************************************************************/
module StoreT(qreg I[r], quNode T[R], quNode Td, int r, int n, quIntr idxIJ[R],
        qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;

        // depreciated
        // int i = quIntrToInt(I);

        for(j = 0; j < R; j++)
        {
                CompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
```

```
                        for(k = 0; k < n; k++)
                        {
                                Toffoli(T[j].reg[k], Td.reg[k], ctrNode[0]);
                        }
                InvCompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);

        }
}

/************************************************************
*       Algorithm 9-1
*       Para: i quInt6
*             T an array of Rbar quIntr
*             Td  r qubits
*             rbar integer (given)
*             r integer (given )
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*************************************************************/
module StoreTrbarr(qreg I[rbar], quIntr T[Rbar], qreq Td[r], int rbar, int r,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;
        // depreciated
        // int i = quIntr6ToInt(I);

        for(j = 0; j < Rbar; j++)
        {
                CompareIJrbar(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
                        for(k = 0; k < r; k++)
                        {
                                Toffoli(T[j].reg[k], Td[k], ctrNode[0]);
                        }
                InvCompareIJrbar(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
        }
}


/************************************************************
*       Algorithm 10
*       Para: i quInt
*             T an array of R quNodes
*             Td a quNode register (that is n qubits)
*             rbar integer (given)
*             b integer (given) [cannot use the global variable n]
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*       Comment: As seen in later function calls, the parameter b
*       could be of values other than n or r.
*************************************************************/
module FetchStoreT(qreg I[rbar], qreg T[Rbar], qreg Td[1], int rbar, int b,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;
        // depreciated
        // and typo in the method       int i = quIntr6ToInt(I);

        for(j = 0; j < Rbar; j++)
        {
                CompareIJrbar(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
                        for(k = 0; k < b; k++)
                        {
                                // Here it is only one bit, hence, do not use SWAP but Swap
                                // Throughout the whole program, when this module is summoned, the parameter b is always 1
                                // Since b is always 1, k is always 0
                                ctrSwap(T[j], Td[k], ctrNode[0]);
                        }
                InvCompareIJrbar(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
        }
}


/**************************************************************
*       Algorithm 11
*       Para: i quInt
*             E qreg[] of size (R(R-1)/2)
*             Ed qreg[] of size R
*             a integer (given)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
**************************************************************/
module FetchE(qreg I[r], qreg E[RR1], qreg Ed[R], int r, quIntr idxIJ[R],
        qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;

        // depreciated
        // int i = quIntrToInt(I);
        int k = 0;
        int kj = 0;
        int jk = 0;

        for(j = 0; j < R; j++)
        {
                CompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
                        for(k = 0; k < j; k++)
                        {
                                kj = ((j*(j-1))/2) + k;
                                Toffoli(Ed[k], E[kj], ctrNode[0]);
                        }

                        for(k = j+1; k < R ; k++)
                        {
                                jk = ((k*(k-1))/2) + j;
                                Toffoli(Ed[k], E[jk], ctrNode[0]);
                        }
                InvCompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
        }
}



/**************************************************************
*       Algorithm 12
*       Para: I quInt of size r
*             E qreg[] of size (R(R-1)/2)
*             Ed qreg[] of size R
*             a integer (given)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
```

```
*************************************************************/
module FetchStoreE(qreg I[r], qreg E[RR1], qreg Ed[R], int r,
        quIntr idxIJ[R], qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j = 0;
        int k = 0;
        int kj = 0;
        int jk = 0;
        // depreciated
        //int i = quIntrToInt(I);

        for(j = 0; j < R; j++)
        {
                CompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode);
                        for(k = 0; k < j; k++)
                        {
                                kj = ((j*(j-1))/2) + k;
                                //Note: only 1 bit change, should be Swap, not SWAP(given in GFI document)
                                // depreciated; because of ctrNode; Swap(E[kj], Ed[k]);
                                ctrSwap(E[kj], Ed[k], ctrNode[0]);
                        }

                        for(k = j+1; k < R ; k++)
                        {
                                // Maybe we can reuse the variable kj without using variable jk
                                jk = ((k*(k-1))/2) + j;
                                //Note: only 1 bit change, should be Swap, not SWAP(given in GFI document)
                                // depreciated; because of ctrNode; Swap(E[jk], Ed[k]);
                                ctrSwap(E[jk], Ed[k], ctrNode[0]);
                        }
                InvCompareIJ(I, idxIJ[j], ctrbit2, ctrbit, ctrNode[0]);
        }
}


/************************************************************
*       Algorithm 13
*       Para: T quNode[R] register
*             Td a quNode: qreg[] of size n
*             Ed qreg[] of size R
*             r integer (given)
************************************************************/

module UPDATE(quNode T[R], quNode Td, qreg Ed[R], int r )
{
        int j = 0;
        qreg tmp[1];

        for(j = 0; j < R; j++)
        {
                EdgeORACLE(T[j], Td, tmp[0]);
                Swap(Ed[j], tmp[0]);
        }
}


/************************************************************
*       Algorithm 14
*       Para: Td a quNode: qreg[] of size n
*             v a quNode: qreg[] of size n
*             n integer (given) [OK to use global variable]
************************************************************/

module SWAP(quNode Td, quNode v, int n )
{
        int j = 0;
        for(j = 0; j < n-1; j++)
        {
                // three CNOT is a simple bit-wise Swap
                CNOT(Td.reg[j], v.reg[j]);
                CNOT(v.reg[j], Td.reg[j]);
                CNOT(Td.reg[j], v.reg[j]);
        }
}


/*****************************************************************************************
*       Algorithm 15
*       Para: T: a quNode[R] register
*                 E: qreg[(R(R-1))/2]
*                 w: a quNode: qreg[] of size n
*                 n integer (given)
*                 r integer (given) [OK to use global r]
*                 GCQWRegs (Graph Collision Workspace Registers)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*       Comment: triTestT set to 1 is T contains the triangle from G
*                        triTestTw set to 1 when w is a node of the triangle with the other edge in T
*****************************************************************************************/

module TestTriangleEdges(quNode T[R], qreg E[RR1], quNode w, int n, int r, GCQWRegs GC,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        // set triTestT to 1 if T contains the triangle
        TriangleTestT(E, n, r, GC.triTestT);


        // Grover with Graph collision
        // Mistake in the GFI document; TriangleEdgeSearch function does not need
        // triTestT parameter as it is embeded in GC register; here I removed that
        // parameter from the code
        TriangleEdgeSearch(T,E,w,n,r,rbar, GC, idxIJ, ctrbit2, ctrbit, ctrNode[0]);


        // Marking T, w and copy that value
        // The data structure of GCQWRegs misses the field "iota" in statement of algorithm 15
        TriangleTestTw(T,E,w, n,r, GC.triTestTw);
}

/************************************************************
*       Algorithm 16
*       Para: E: qreg[(R(R-1))/2]
*             n integer (given)
*                r integer (given) [OK to use the global r]
*                test: qreg[1]
************************************************************/
module TriangleTestT(qreg E[RR1], int n, int r, qreg test[1])
{
        int i = 0;
        int j = 0;
        int k = 0;
        int ij = 0;
        int ik = 0;
        int jk = 0;

        for(i=0; i < R; i ++)
        {
                for (j = i+1; j < R; j++)
                {
                        ij = i + (j*(j-1))/2;
                                for (k = j+1; k < R; k++)
                                {
                                        ik = i + (k*(k-1))/2;
                                        jk = j + (k*(k-1))/2;
                                        Toffoli(ctrbit[0], E[ij], E[ik]);
                                        Toffoli(ctrbit[1], ctrbit[0], E[jk]);
                                        EvaluateOR(ctrbit[2], ctrbit[1], test[0]);
                                        CNOT(ctrbit[3], ctrbit[2]);

                                        // uncompute
                                        InvEvaluateOR(ctrbit[2], ctrbit[1], test[0]);
                                        Toffoli(ctrbit[1], ctrbit[0], E[jk]);
                                        Toffoli(ctrbit[0], E[ij], E[ik]);

                                        ctrONE(test[0], ctrbit[3]);

                                        // need ancilla here
                                        ZERO(ctrbit[3]);
                                }
                }
        }
}


/***************************************************************
*       Algorithm 17
*       Para: T: a quNode[R] register
*                 E: qreg[(R(R-1))/2]
*                 w: a quNode: qreg[] of size n
*             n: integer (given)
*                 r: integer (given)
*                 test: qreg[1]
***************************************************************/
module TriangleTestTw(quNode T[R], qreg E[RR1], quNode w, int n, int r, qreg test[1])
{
        qreg Ed[R];
        qreg tmp[R]
        int i = 0;
        int j = 0;
        int ij = 0;

        ZERO(test);

        for(i = 0; i < R; i++)
        {
                // Store value into Ed register
                EdgeORACLE(T[i], w, tmp[i]);
                CNOT(Ed[i], tmp[i]);
        }

        for(i = 0; i < R; i++)
        {
                for (j = i+1; j < R; j++)
                {
                        ij = i + (j*(j+1))/2;
                        Toffoli(ctrbit[0], Ed[i], Ed[j]);
                        Toffoli(ctrbit[1], ctrbit[0], E[ij]);
                        EvaluateOR(ctrbit[2], ctrbit[1], test[0]);
                        CNOT(ctrbit[3], ctrbit[2]);

                        // uncompute
                        InvEvaluateOR(ctrbit[2], ctrbit[1], test[0]);
                        Toffoli(ctrbit[1], ctrbit[0], E[ij]);
                        Toffoli(ctrbit[0], Ed[i], Ed[j]);

                        ctrONE(test[0], ctrbit[3]);

                        // need ancilla here
                        ZERO(ctrbit[3]);
                }
        }

        for(i = 0; i < R; i++)
        {
                // Uncompute Ed register
                CNOT(Ed[i], tmp[i]);
        }
}



/**************************************************************
*       Algorithm 18
*       Para: T: a quNode[R] register
*                 E: qreg[(R(R-1))/2]
*                 w: a quNode: qreg[] of size n
*             n: integer (given)
*                 r: integer (given)
                  rbar: integer (given)
*                 GCQWRegs (Graph Collision Workspace Registers)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
**************************************************************/
module TriangleEdgeSearch(quNode T[R], qreg E[RR1], quNode w, int n, int r, int rbar, GCQWREGS GC,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int i = 0;

        // Grover Search on w using graph collision to T, w
        for(i = 0; i < n; i++)
        {
                H(w.reg[i]);
        }

        for(i = 0; i < tg; i++)
        {
                GCQWalk(T,E, w, n,r, rbar, GCQWRegs, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                ZeroRegTest(GC.ctri, ctrbit, CTR);
                X(ctrbit[CTR-2]);
                Toffoli(ctrbit[CTR-1], ctrbit[CTR-2], GC.triTestT);
                CNOT(w.phase[0], ctrbit[CTR-1]);

                // uncompute
                Toffoli(ctrbit[CTR-1], ctrbit[CTR-2], GC.triTestT);
                X(ctrbit[CTR-2]);
                ZeroRegTest(GC.ctri, ctrbit, CTR);

                //TODO: Assume the inverse function will be present when provided the original circuit
                InvQCQWalk(T,E, w, n, r, rbar, GC, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                // Diffuse on the node
                DIFFUSENode(w, n);

        }
}
```

```
/*************************************************************
*       Algorithm 19
*       Para: T: a quNode[R] register
*                 E: qreg[(R(R-1))/2]
*                 w: a quNode: qreg[] of size n
*             n: integer (given)
*                 r: integer (given)
*                 rbar: integer (given)
*                 GCQWRegs (Graph Collision Workspace Registers)
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*************************************************************/
module GCQWalk(quNode T[R], qreg E[RR1], quNode w, int n, int r, int rbar, GCQWREGS GC,
        quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        quNode Td;
        qreg Ed[R];
        qreg taud[r];
        qreg Ewd;
        qreg Edd[Rbar];

        int j = 0;
        int k = 0;

        for(j = 0; j < Rbar ; j++)
        {
                for(k = 0; k < r; k++)
                {
                        H((GC.tau[j]).reg[k]);
                }
        }

        //Apply Hadamard to Walk Registers
        for(j = 0; j < rbar ; j++)
        {
                H(GC.iota[j]);
        }

        for(j = 0; j < r ; j++)
        {
                H(GC.sigma[j]);
        }


        // Zero the workspace registers
        for(j = 0; j < n; j++)
        {ZERO(Td.reg[j]); }

        for(j = 0; j < R; j++)
        {ZERO(Ed[j]); }

        for(j = 0; j < Rbar; j++)
        {ZERO(GC.Ew[j]); }

        for(j = 0; j < 2*rbar-1; j++)
        {ZERO(GC.cTri[j]); }

        for(j = 0; j < r; j++)
        { ZERO(taud); }

        ZERO(Ewd);
        for(j = 0; j <Rbar; j++)
        { ZERO(Edd[j]); }

        //Note: In GFI document, it did extra ZERO on the following. It was done earlier already in this procedure.
        //ZERO(Td, n);
        //ZERO(Ed, R);


        //Set up the walk state
        for(j = 0; j < Rbar; j++)
        {
                FetchT(GC.tau[j], T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                // can directly write result into Ew since it is initialized in 0
                EdgeORACLE(Td, w, GC.Ew[j]);
                FetchT(GC.tau[j], T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
        }

        for(j=0; j < Rbar; j++)
        {
                FetchE(GC.tau[j], E, Ed, r, ctrbit2, ctrbit, ctrNode[0]);
                for(k = j+1; k < Rbar; k++)
                {
                        FetchTr1(GC.tau[k], Ed, Edd[k], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

                        Toffoli(ctrbit[0], GC.Ew[j], GC.Ew[k]);
                        Toffoil(ctrbit[1], ctrbit[0], Edd[k]);

                        ctrCTRincr(GC.cTri, ctrbit[1], ctrbit2, ctrNode[0]);

                        // uncompute
                        Toffoil(ctrbit[1], ctrbit[0], Edd[k]);
                        Toffoli(ctrbit[0], GC.Ew[j], GC.Ew[k]);

                        FetchTr1(GC.tau[k], Ed, Edd[k], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
                }
                FetchE(GC.tau[j], E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
        }

        //Execute the Graph Collision Random Walk
        for(j = 0; j < tbarm; j++)
        {
                ZeroRegTest(GC.cTri, ctrbit, CTR);
                X(ctrbit[CTR-2]);
                X(GC.triTestT);
                Toffoli(ctrbit[CTR-1], ctrbit[CTR-2], GC.triTestT);

                //phase flip and the phase bit is always stored at the Rbar_th element's first qubit
                CNOT((GC.tau[Rbar]).reg[0], ctrbit[CTR-1]);

                //uncompute
                Toffoli(ctrbit[CTR-1], ctrbit[CTR-2], GC.triTestT);
                X(GC.triTestT);
                X(ctrbit[CTR-2]);
                ZeroRegTest(GC.cTri, ctrbit, CTR);


                for(k = 0; k<tbarw; k++)
                {
                        GCQWalkStep(T,E, w, n, r, rbar, GC, Td, Ed, taud, Ewd, Edd, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
                }
        }
}
```

```
/************************************************************
*       Algorithm 20
*       Para: T: a quNode[R] register
*                 E: qreg[(R(R-1))/2]
*                 w: a quNode: qreg[] of size n
*             n: integer (given)
*                 r: integer (given)
*                 rbar: integer (given)
*                 GCQWRegs (Graph Collision Workspace Registers)
*                 Td: quNode (that is qreq[n])
*                 Ed: qreg[R]
*                 taud: qreg[r]
*                 Ewd:  qreg
*                 Edd:  qreg[Rbar]
*                 quIntr idxIJ[R]
*                 qreg ctrbit2[n-1]
*                 qreg ctrbit[n-1]
*             qreg ctrNode[1]
*************************************************************/

module GCQWalkStep(quNode T[R], qreg E[RR1], quNode w, int n, int r, int rbar, GCQWREGS GC, quNode Td, qreg Ed[R],
        qreg taud[r], qreg Ewd[1], qreq Edd[Rbar], quIntr idxIJ[R],qreg ctrbit2[n-1], qreg ctrbit[n-1], qreg ctrNode[1])
{
        int j;
        qreg tmp[1];

        //Perform the Diffuse function on iota and sigma
        // + 1 because we have extra phase bit
        DIFFUSEGC(GC.iota , GC.sigma, rbar+r+1);

        // Prepare data register for SWAP and Update
        //Computing taud
        FetchTrbarr(GC.iota, GC.tau, taud, rbar, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        //Computig Td
        FetchT(taud, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);


        FetchStoreT(GC.iota, GC.Ew, Ewd, rbar, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        //Computing Ed
        FetchE(taud, E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

        // Fetching and computing Edd
        for(j = 0; j < Rbar; j++)
        {
                FetchTr1(GC.tau[j], Ed, Edd[j], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
        }


        // Update GC.cTri using current edge values in Ewd
        for(j = 0; j < Rbar; j++)
        {
                Toffoli(ctrbit[0], Ewd, GC.Ew[j]);
                Toffoli(ctrbit[1], ctrbit[0], Edd[j]);

                ctrCTRdecr(GC.cTRi, ctrbit[1], ctrbit2, ctrNode[0]);

                //uncompute
                Toffoli(ctrbit[1], ctrbit[0], Edd[j]);
                Toffoli(ctrbit[0], Ewd, GC.Ew[j]);
        }


        // Erase current edge values in Ewd
```

```
EdgeORACLE(Td, w, tmp[0]);
CNOT(Ewd, tmp[0]);

// Prepare data register for SWAP
for(j = 0; j < Rbar; j++)
{       // Uncompute Edd
        FetchTr1(GC.tau[j], Ed, Edd[j], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
}

FetchE(taud, E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);                                                // Uncompute Ed

FetchT(taud, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);                                             // Uncompute Td

StoreTrbarr(GC.iota, GC.tau, taud, rbar, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);


for(j = 0; j < r; j++)
{
        Swap(taud[j], GC.sigma[j]);
}

// Uncompute data registers after SWAP operation
StoreTrbarr(GC.iota, GC.tau, taud, rbar, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

FetchT(taud, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

FetchE(taud, E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

for(j = 0; j < Rbar; j++)
{
        FetchTr1(GC.tau[j], Ed, Edd[j], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
}

// Compute new edge info in Ewd
// Can directly call tmp[0] as Oracle writes the right value despite of the initial value of tmp[0]
EdgeORACLE(Td,w, tmp[0]);
CNOT(Ewd, tmp[0]);

// Update Ewd and GC.cTri using  any new triangles involving Ewd
for(j = 0; j < Rbar; j++)
{
        Toffoli(ctrbit[0], Ewd, GC.Ew[j]);
        Toffoli(ctrbit[1], ctrbit[0], Edd[j]);

        ctrCTRdecr(GC.cTRi, ctrbit[1], ctrbit2, ctrNode[0]);

        //uncompute
        Toffoli(ctrbit[1], ctrbit[0], Edd[j]);
        Toffoli(ctrbit[0], Ewd, GC.Ew[j]);
}

// Restore data and data registers to initial state
for(j  = 0; j < Rbar; j++)
{
        FetchTr1(GC.tau[j], Ed, Edd[j], r, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
}


// Could use ZERO as this step is to set Ed to 0
FetchE(taud, E, Ed, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
FetchStoreT(GC.iota, Ew, Ewd, rbar, 1, idxIJ, ctrbit2, ctrbit, ctrNode[0]);

// Could use ZERO as this step is to set Td to 0
FetchT(taud, T, Td, r, n, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
```

```
        // Could use ZERO as this step is to set taud to 0
        FetchTrbarr(GC.iota, GC.tau, taud, rbar, r, idxIJ, ctrbit2, ctrbit, ctrNode[0]);
}

/**************************************************************
*       Algorithm Testing for Main
***************************************************************/
module main()
{
        int i;
        int b=4;
        int j;

    QWTFP(15,9);

}
```

B Quantum Resources Count for Individual Module

In this section, we provide the quantum resource estimate for each module (without reusing the ancillae) that is used in the program. As mentioned in section 2.2, we have the overall structure layout. But that structure (with the $t_m, t_w, t_g, tbar_m, tbar_w$ iterations) is only good for obtaining the upper bound for the total number of oracle queries. To obtain the quantum resources estimate, we need to know the resource cost for each individual module.

We summarize the given parameters and their relations in table 10 here again. They would be useful for computing the required logical gates and logical qubits for the program QWTFP [10].

variable	n	r	N	R	rbar	Rbar	CTR	t_m	t_w	$tbar_m$	$tbar_w$	t_g
value	15	9	32768	512	6	64	11	64	22	8	22	142
Relations			2^n	2^r	$\frac{2r}{3}$	2^{rbar}	$2 * rbar - 1$	N/R	$\sqrt{R}$	$R/Rbar$	$\sqrt{R}$	$\frac{\pi\sqrt{N}}{4}$

Table 10: The Variables and Their Values

We suggest that the reader should read along with the source code of the corresponding module in Appendix A for this section. The result might be presented in the following vector format $(CNOT, H, S, T, Tdag, X, Qubit, Measure, Z)$. For instance $(2, 3, 0, 0, 0, 0, 0, 0, 0)$ means it uses 2 CNOT gates and 3 Hadamard gates. We also include the script that was used for counting the resources and epoches in this appendix.

Swap
= 3 CNOT

ctrSwap
= 3 Toffoli

Toffoli
= 6 CNOT + 2 H + 1 S + 3 T + 4 Tdag

ONE
= 2 CNOT + 1 X + 1 qubit

ctrONE
= 1 CNOT + 2 Toffoli + 1 qubit
= 13 CNOT + 4 H + 2 S + 6 T + 8 Tdag + 1 qubit

ZERO
= 2 CNOT + 1 qubit

INITIALIZE
= 1 ZERO + 1 H
= 2 CNOT + 1 H+ 1 qubit

QWTFP: Initializing Registers[11]
= (2 + n + r(Rbar) + rbar + r + Rbar + (2 × rbar-1) + 2) ZERO + (nR+ r + n) INITIALIZE

[10]Here we solely look at the resources required by QWTFP and consider the Oracle module as 1 unit.
[11]We follow the pseudocode and keep the ZERO operation.

+ 4 (H + X)(set up phase bits) + (R $\times$ r) X (set up index)
= 16778 CNOT + 7708 H + 4612 X + 8389 qubit

SETUP[12]:
= (R(R-1)/2) ZERO + (R(R-1)/2) ZERO + (R(R-1)/2) Oracle calls
= 523264 CNOT + 261632 qubit + 130816 Oracle calls

QWTFP: Final measuring when there is a triangle
= EMeasure + TMeasure + nodeMeasure
= (R(R-1)/2 Measure) + (Rn Measure) + n Measure
= 138511 Measure

QWTFP: Getting a minus sign for the state
= 2 EvaluateOR + 1 CNOT
= 2 Toffoli + 6 X + 1 CNOT
= 13 CNOT + 4 H + 2 S + 6 T + 8 Tdag + 6 X

EvaluateOR (same for the InvEvaluateOR)
= 3 X + 1 Toffoli
= 6 CNOT + 2 H + 1 S + 3 T + 4 Tdag + 3 X

ZeroNodeTest (same for the InvZeroNodeTest)
= n X + (n-1) Toffoli
= 6 (n-1) CNOT + 2 (n-1) H + (n-1) S + 3 (n-1) T + 4 (n-1) Tdag + n X

ZeroRegTest (same for the InvZeroRegTest): with parameter integer k
= k X + (k-1) Toffoli
= 6 (k-1) CNOT + 2 (k-1) H + (k-1) S + 3 (k-1) T + 4 (k-1) Tdag + k X

CompareIndexr
= 2r CNOT

CompareIndexrbar
= 2rbar CNOT

OneRegTest[13]
= CNOT + (n-1) Toffoli

CompareIJ
= CompareIndexr + OneRegTest

CompareIJrbar
= CompareIndexrbar + OneRegTest

ctrCTRincr
= OneRegTest + (2rbar-1) CNOT + (n+1) ZERO

ctrCTRdecr
= ZeroNodeTest + (2rbar-1) CNOT + (n+1) ZERO + (2rbar-1) X

[12]We follow the pseudocode and keep the ZERO operation.
[13]It varies with parameter k ranging from 6 to n; be pessimistic, let us choose n.

DIFFUSENODE
= 2n H + X + 2 ZeroNodeTest + 1 CNOT
= 2n H + X + (2n X + 2(n-1) Toffoli) + 1 CNOT
= 2n H + (2n+1) X + 2(n-1) Toffoli + 1 CNOT
= 169 CNOT + 86 H + 28 S + 84 T + 112 Tdag + 31 X

DIFFUSEVI
= 2(n+r) H + X + 2(ZeroNodeTest + ZeroRegTest(parameter r)) + 2 Toffoli + 1 CNOT
= 2(n+r) H + X + (2(n+r) X + 2(n+r-2) Toffoli + 2 Toffoli) + 1 CNOT
= 2(n+r) H + (2(n+r)+1) X + 2(n+r-1) Toffoli + 1 CNOT
= 277 CNOT + 140 H + 46 S + 138 T + 184 Tdag + 49 X

DIFFUSEGC
= 2(rbar+r) H + X + 2(ZeroRegTest(parameter rbar) + ZeroRegTest(parameter r)) + 2 Toffoli + 1 CNOT
= 2(rbar+r) H + (2(rbar+r)+1) X + 2(rbar+r-1) Toffoli + 1 CNOT
= 169 CNOT + 86 H + 28 S + 84 T + 112 Tdag + 31 X

FetchT[14]
= 15R Toffoli + 2R CompareIJ
$= (151552, 44032, 22016, 66048, 88064, 0, 0, 0, 0)$

FetchE
= R× R Toffoli + 2R CompareIJ
$= (1678336, 552960, 276480, 829440, 1105920, 0, 0, 0, 0)$

StoreT[15]
=15R Toffoli + 2R CompareIJ
$= (151552, 44032, 22016, 66048, 88064, 0, 0, 0, 0)$

FetchStoreT
= Rbar ctrSwap + 2Rbar CompareIJrbar
$= (13568, 3968, 1984, 5952, 7936, 0, 0, 0, 0)$

FetchStoreE
= 2R CompareIJ + R × R ctrSwap;
$= (4824064, 1601536, 800768, 2402304, 3203072, 0, 0, 0, 0)$

UPDATE
= 2^r Swap + 2^r Oracle calls + 1 qubit
= 1536 CNOT + 512 Oracle calls + 1 qubit

SWAP[16]
= $3n$ CNOT = 45 CNOT

QWSH
= (R+n) qubits + 2n ZERO + DIFFUSEVI + 2 FetchT + 2 FetchStoreE

[14]There are 3 types of FetchT (FetchT, FetchTr1, FetchTrbarr), the operations are the same but different in the value of the parameter (actually those parameters are close in value). Hence, we chose the largest one, FetchT, to represent.
[15]There are 2 types of StoreT (StoreT, StoreTrbarr), the operations are the same but different in the value of the parameter (actually those parameters are close in value). Hence, we chose the largest one, StoreT, to represent.
[16]Different from Swap, this SWAP operation acts on a quNode, not a qubit.

+ 2 UPDATE + 2 StoreT + n Swap
$= (10257790, 3379340, 1689646, 5068938, 6758584, 49, 559, 0, 0)$ + 1024 Oracle calls

TestTriangleEdges
= TriangleTestT + TriangleEdgeSearch + TriangleTestTw
$\approx$ $(3051293128516, 901931321295, 450964824828, 1352894474484, 1803859299312, 302180354, 45133229, 0, 0)$ + 102184 Oracle

TriangleTestT
$= ((\sum_{i=0}^{R-1} \sum_{j=i+1}^{R-1} \sum_{k=j+1}^{R-1} 1)\times$ (2(2 Toffoli + EvaluateOR) + 1 CNOT + (1 ctrONE + 1 ZERO))
$= \frac{1}{6}(R-2)(R-1)R\times$ [17] (6 Toffoli + 6 X + 1 CNOT + (3 CNOT + 2 Toffoli + 2 qubit))
$= \frac{1}{6}(R-2)(R-1)R\times$ (8 Toffoli + 6 X + 4 CNOT + 2 qubit)
$= (1156413440, 355819520, 177909760, 533729280, 711639040, 133432320, 44477440, 0, 0)$

TriangleTestTw
$= (\sum_{i=0}^{R-1} \sum_{j=i+1}^{R-1} 1)\times$ (2(2 Toffoli + EvaluateOR) + 1 CNOT + (1 ctrONE + 1 ZERO))
+ R oracle calls + 2R CNOT
$= R(R-1)/2\times$ [18] (8 Toffoli + 6 X + 4 CNOT + 2 qubit) + R oracle calls + 2R CNOT
$= (6803456, 2093056, 1046528, 3139584, 4186112, 784896, 261632, 0, 0)$ + 512 Oracle calls

TriangleEdgeSearch
= 15 H + t_g GCQWalk
+ t_g(2 ZeroRegTest(parameter 2rbar -1) + 2 X + 2 Toffoli + 1 CNOT)
+ t_g DIFFUSENODE + t_g InverseGCQWalk [19]
= 15 H + $2t_g$ GCQwalk + t_g (2n H + (2(n+CTR)+3)X + 2(n+CTR-1) Toffoli + 2 CNOT)
$= (3050129911620, 901573408719, 450785868540, 1352357605620, 1803143474160, 167963138, 394157, 0, 0)$ + 101672 Oracle calls

GCQWalk
= (R+n+r+1+Rbar) qubits + (Rbar×r + rbar + r)H + (n + R + Rbar + 2rbar -1 + r + 1 + Rbar) ZERO + rbar (Oracle + 2FetchT) + Rbar FetchE + ((Rbar-1)Rbar/2)(FetchT + 4× Toffoli + 1 ctrCTRincr + FetchT) + Rbar FetchE + $tbar_m$(4X + 2 Toffoli + 2 ZeroRegTest + 1 CNOT) + $tbar_m \times tbar_w$ GCQWalk-Step
$\approx (10739893904, 3174554191, 1587274160, 4761822480, 6349096640, 591392, 394157, 0, 0)$ + 358 Oracle calls

GCQWalkStep
= 1 qubit + 1 DIFFUSEGC + 2 FetchT + 1 FetchStoreT + 1 FetchE + Rbar (FetchT + 4 Toffoli + ctrCTRdecr) + 1 Oracle + 1 CNOT + Rbar FetchT + 1 Fetch T + 1 FetchE + 1 StoreT + r Swap + 1 FetchT + 1 FetchE + Rbar FetchT + 1 Oracle + 1 CNOT + Rbar (FetchT + 4 Toffoli + ctrCTRdecr + FetchT) + 1 FetchE + 1 FetchStoreT + 2 FetchT
$\approx (56317510, 16622934, 8311452, 24934356, 33245808, 3359, 2049, 0, 0)$ + 2 Oracle calls

---

[17] $\sum_{i=0}^{R-1} \sum_{j=i+1}^{R-1} \sum_{k=j+1}^{R-1} 1 = \sum_{t=1}^{R-1} t(R-1-t) = \frac{1}{6}(R-2)(R-1)R = 22238720$

[18] $\sum_{i=0}^{R-1} \sum_{j=i+1}^{R-1} 1 = R(R-1)/2 = 130816$

[19] The inverse should have the same complexity as GCQWalk.

```
(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Resources Counting for Triangle Finding Problem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Definition of the vector: an 1 by 9 vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
V ={CNOT, H, S, T, Tdag, X, Qubit, Measure, Z}
*)

(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear and then declare variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
ClearAll["Global`*"];

tm = 64;
tw = 22;
tbarm = 8;
tbarw = 22;
tg= 142;
n = 15;
r = 9;
rbar = 6;
Rbar = 64;
R=512;
ctr = 11;

RRR = 22238720;
RR = 130816; (* RR = R*(R-1)/2 *)
RRbar = 2016; (* RRbar = Rbar*(Rbar-1)/2 *)
tbarm = 8;
tbarw = 22;
tg=142;

(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define basic building blocks's values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
quBit = {0,0,0,0,0,0,1,0,0};
H =  {0,1,0,0,0,0,0,0,0};
CNOT = {1,0,0,0,0,0,0,0,0};
X = {0,0,0,0,0,1,0,0,0};
M = {0,0,0,0,0,0,0,1,0};

Toffoli = {6,2,1,3,4,0,0,0,0};
EvaluateOR = Toffoli + 3*X;
ONE = {2,0,0,0,0,1,1,0,0};
ZERO = {2,0,0,0,0,0,1,0,0};
INITIALIZE = 2*CNOT + H + quBit;
Swap = 3*CNOT;
ctrSwap = 3*Toffoli;

ctrONE = CNOT + 2*Toffoli;
ZeroNodeTest = (n-1)*Toffoli + n*X;
ZeroRegTestctr = (ctr-1)*Toffoli + ctr*X;
ZeroRegTestr = (r-1)*Toffoli + r*X;
ZeroRegTestrbar = (rbar-1)*Toffoli + rbar*X;


CompareIndexr = 2*r*CNOT;
```

```
CompareIndexrbar = 2*rbar*CNOT;

(* varies with parameter k ranging from 6 to n; be pessimistic, let us choose n *)
OneRegTest = CNOT + (n-1)*Toffoli;

CompareIJ = CompareIndexr + OneRegTest;
CompareIJrbar = CompareIndexrbar + OneRegTest;

ctrCTRincr = OneRegTest + (2*rbar-1)*CNOT + (n+1)*ZERO;
(* use ZeroNodeTest to approximate ZeroRegTest *)
ctrCTRdecr = ZeroNodeTest + (2*rbar-1)*CNOT + (n+1)*ZERO + (2*rbar-1)*X;


INITIALIZE = 2*CNOT + H + quBit;
FetchT = R*15*Toffoli + 2*R*CompareIJ;
FetchE = R*R*Toffoli + 2*R*CompareIJ;
StoreT = R*15*Toffoli + 2*R*CompareIJ;
FetchStoreT = Rbar*ctrSwap + 2*Rbar*CompareIJrbar;
FetchStoreE = 2*R*CompareIJ + R*R*ctrSwap;
theUpdate = 3*R*CNOT + quBit;


DIFFUSENODE = {169, 86, 28, 84, 112, 31, 0,0,0};
DIFFUSEVI = {277,140,46,138,184,49,0,0,0};
DIFFUSEGC = {169, 86, 28, 84, 112, 31, 0,0,0};


(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Setup, initialization pluse QWSH
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
RI = (2+n+r*Rbar + rbar + r + Rbar + (2*rbar-1)+2)*ZERO +  (n*R+ r + n)*INITIALIZE + 4*(H + X) + R*r*X;
QWSH = (R+n)*quBit + 2*n*ZERO + DIFFUSEVI + 2*FetchT + 2*FetchStoreE + 2*theUpdate + 2*StoreT + n*Swap;
SU = 2*RR*ZERO;


(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The final measures
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
Mea=((R*(R-1))/2 + R*n + n){0,0,0,0,0,0,0,1,0};

(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the cost for those larger modules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
GCQWalkStep = {0,0,0,0,0,0,1,0,0} + DIFFUSEGC + 2*FetchT + FetchStoreT + FetchE;
GCQWalkStep = GCQWalkStep + Rbar*(FetchT + 4*Toffoli + ctrCTRdecr)+ {1,0,0,0,0,0,0,0,0};
GCQWalkStep = GCQWalkStep + Rbar*FetchT + FetchT + FetchE + StoreT + r*Swap + FetchT + FetchE;
GCQWalkStep = GCQWalkStep + Rbar*FetchT + {1,0,0,0,0,0,0,0,0} + Rbar*(FetchT + 4*Toffoli + ctrCTRdecr + FetchT);
GCQWalkStep = GCQWalkStep + FetchE + FetchStoreT + 2*FetchT;

GCQWalk = (R+n+r+1+Rbar)*quBit + (Rbar*r + rbar + r)*H + (n + R + Rbar + ctr + r + 1 + Rbar)*ZERO+ 2*rbar*FetchT;
GCQWalk = GCQWalk + + Rbar*FetchE + (((Rbar-1)*Rbar)/2)*(FetchT + 4*Toffoli + ctrCTRincr + FetchT) + Rbar*FetchE;
GCQWalk = GCQWalk + tbarm*(4*X + 2*Toffoli + 2*ZeroRegTestctr + CNOT) + tbarm*tbarw*GCQWalkStep;

TriangleEdgeSearch = 15*H + 2*tg*GCQWalk + tg*(2*n*H + (2*(n+ctr)+3)*X + 2*(n+ctr-1)*Toffoli + 2*CNOT);

(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set the qubit to GCQWalk[[7]] because of reuse
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
TriangleEdgeSearch[[7]] = GCQWalk[[7]];


TriangleTestT = (((R-2)*(R-1)*R)/6)*(8*Toffoli + 6*X + 4*CNOT + 2*quBit);
TriangleTestTw = ((R*(R-1))/2)*(8*Toffoli + 6*X + 4*CNOT + 2*quBit) + 2*R*CNOT;
TestTriangleEdges = TriangleEdgeSearch + TriangleTestTw + TriangleTestT;

(* Notice that for qubits we should not multiply by 2tim + 1  and tm tw since they can be reused *)
QWTFP = RI + SU + (2*tm+1)*TestTriangleEdges + tm*tw*QWSH + Mea;
QWTFP[[7]] = QWTFP[[7]] -  (2*tm)*TestTriangleEdges[[7]];

(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute in terms of epochs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
epoch3 = (2*tm+1)*TestTriangleEdges + tw*tm*QWSH;
epoch3sum = epoch3[[1]]+ epoch3[[2]]+epoch3[[3]]+epoch3[[4]]+epoch3[[5]]+epoch3[[6]]+epoch3[[8]];
epoch3percent = epoch3/epoch3sum;
```

C Alternative to Module 16 and Module 17

In this section, we include a simple alternative for Algorithm 16 and Algorithm 17 summoned by the QWTFP module.

```
/**************************************************************************************
* File: TriangleFindingProblem.quantum
*
* Author: Chen-Fu Chiang @ USherbrooke
*
* Implementation of the triangle finding algorithm specified in the
* Quantum Computer Science Program @ Government Furnished Information document
*
* Document date: June 20, 2011
* Document Version: 1.0.0
* Document Reversion number: 188
*
* Implementation date: 2012
**************************************************************************************/
module ctrEvaluate16(qreg dum[3],qreg ctr4[1], qreg ctr3[1], qreg ctr2[1], qreg ctr1[1])
{
        Toffoli(dum[0], ctr2[0], ctr1[0]);
        Toffoli(dum[1], ctr3[0], dum[0]);
        Toffoli(dum[2], ctr[4], dum[1]);

        // partial uncompute
        Toffoli(dum[1], ctr3[0], dum[0]);
        Toffoli(dum[0], ctr2[0], ctr1[0]);
}

/***********************************************************
*       Algorithm 16
*       Para: E: qreg[(R(R-1))/2]
*             n integer (given)
*                r integer (given) [OK to use the global r]
*                test: qreg[1]
***********************************************************/
module TriangleTestT(qreg E[RR1], int n, int r, qreg test[1])
{
        int i = 0;
        int j = 0;
        int k = 0;
        int ij = 0;
        int ik = 0;
        int jk = 0;
        qreg dum[3];

        for(i=0; i < R; i ++)
        {
                for (j = i+1; j < R; j++)
                {
                        ij = i + (j*(j-1))/2;
                                for (k = j+1; k < R; k++)
                                {
                                        ik = i + (k*(k-1))/2;
                                        jk = j + (k*(k-1))/2;

                                        // Here it is a bit tricky as once we get 1 for test,
                                        // we don't care what happen to
                                        // the qubit at dummy register[2] as we will not execute
                                        // this ctrEvaluate16 again. And by doing so, we can avoid
                                        // many ancillae!

                                        // when test[0] = 1, we don't  do anything
                                        // when test[0] = 0, the only time test[0] could be
                                        // changed is when  E[jk], E[ij], E[ik] = 1
                                        X(test[0]);

                                        // since we do partial uncompute for dum[0], dum[1];
                                        // we are in good hand; and if  E[jk] AND E[ij] AND E[ik]  != 1
                                        // then dum[2] is always 0; then the dum register is
                                        // still good for next iteration
                                        ctrEvaluate16(dum, E[jk], E[ij], E[ik], test[0]);

                                        X(test[0]);

                                        // when dum[2] = 1, it is only the case when test[0] = 0, and
                                        // E[jk] AND E[ij] AND E[ik] = 1
                                        CNOT(test[0], dum[2]);

                                        // once test[0] is one, in next iteration, X(test[0])
                                        // is always 0, ctrEvaluate16(dum, E[jk], E[ij], E[ik], test[0])
                                        // would never be executed


                                }
                        }
                }
        }


/************************************************************
*       Algorithm 17
*       Para: T: a quNode[R] register
*               E: qreg[(R(R-1))/2]
*               w: a quNode: qreg[] of size n
*             n: integer (given)
*               r: integer (given)
*               test: qreg[1]
*************************************************************/
module TriangleTestTw(quNode T[R], qreg E[RR1], quNode w, int n, int r, qreg test[1])
{
        qreg Ed[R];
        qreg tmp[R]
        int i = 0;
        int j = 0;
        int ij = 0;
        qreg dum[3];

        ZERO(test);

        for(i = 0; i < R; i++)
        {
                // Store value into Ed register
                EdgeORACLE(T[i], w, tmp[i]);
                CNOT(Ed[i], tmp[i]);
        }

        for(i = 0; i < R; i++)
        {
                for (j = i+1; j < R; j++)
                {
                        ij = i + (j*(j+1))/2;

                                        // Same reasoning as given in module 16
                                        X(test[0]);
                                        ctrEvaluate16(dum, E[ij], Ed[j], Ef[i], test[0]);
                                        X(test[0]);
                                        CNOT(test[0], dum[2]);
                }
        }

        for(i = 0; i < R; i++)
        {
                // Uncompute Ed register
                CNOT(Ed[i], tmp[i]);
        }
}
```

D Parallel Factor Count

Here we count the numbers of parallel and sequential operations (for CNOT, H, X, Z, Measure) inside each module. We will use the following notations:

- A-B-PL-G and C-SQ-G

where A, B and are positive integers and G are the corresponding gate, such as H, X, Z and M (Measure). PL stands for parallel and SQ stands for sequential. For instance, 3-15-PL-H means that 15 Hadamard gates act on 15 qubits in parallel and this action repeats 3 times sequentially. Similarly, 2-SQ-X means bit flip is performed on 1 qubit and it is repeated twice sequentially. At the end of the major modules, we will compute then list the occurrence of the elementary gates in the following format:

- G-D-E or G(D,E)

where D and E are positive integers and G is the gate. For instance, if we have 2-15-PL-H and 10-SQ-H in the module, then we know the occurrence of H gate for this module is H-12-40 as H gate appears at 12 different time intervals and the total H gate count is 40. Hence, the parallelization factor of H gate in this module is 40/12.

We suggest that the reader should read along with the source code of the corresponding module in Appendix A and the analysis given in Appendix B for this section. We also include the script that was used for counting the parallel factor in this appendix.

Swap
= 3-SQ-CNOT
= CNOT-3-3

ctrSwap
= 3-SQ-Toffoli
= Toffoli-3-3

ONE
= 2-SQ-CNOT + 1-SQ-X
= CNOT-2-2 + X-1-1

ctrONE
= 1-SQ-CNOT + 2-SQ-Toffoli
= Toffoli-2-2 + CNOT-1-1

ZERO
= 2-SQ-CNOT
= CNOT-2-2

INITIALIZE
= 2-SQ-CNOT + 1-SQ-H
= CNOT-2-2 + H-1-1

QWTFP: Initializing Registers[20]
= (2 + n + r(Rbar) + rbar + r + Rbar + (2 × rbar-1) + 2) ZERO + (nR+ r + n) INITIALIZE + Rr X
= 2-8389-PL-CNOT + 1-7708-PL-H + 1-4-PL-X + 1-4608-PL-X

[20]We follow the pseudocode and keep the ZERO operation.

= CNOT-2-16778 + H-1-7708 + X-2-4612

SETUP[21]:
= (R(R-1)/2) ZERO + (R(R-1)/2) ZERO
= 4-130816-PL-CNOT
= CNOT-4-523264

QWTFP: Final measuring when there is a triangle
EMeasure = M-1-130816
TMeasure = M-1-7620
nodeMeasure = M-1-15

EvaluateOR (same for InvEvaluateOR)
= 1-2-PL-X + 1-SQ-Toffoli + 1-SQ-X
= Toffoli-1-1 + X-2-3

CompareIndexr
= CNOT-2-2r

OneRegTest
=(Toffoli-(n-1)-(n-1) + CNOT-1-1

CompareIJ
= CompareIndexr-1-1 + OneRegTest-1-1

CompareIndexrbar
= CNOT-2-2rbar

CompareIJrbar
= CompareIndexrbar-1-1 + OneRegTest-1-1

Toffoli
= CNOT-6-6 + H-2-2 + S-1-1 + T-3-3 + Tdag-4-4

QWTFP: Getting a minus sign for the state
= 2-SQ-EvaluateOR + 1-SQ-CNOT
= Toffoli-2-2 + X-2-4 + CNOT-1-1

ZeroNodeTest (same for InvZeroNodeTest)
= 1-n-PL-X + (n-1)-SQ-Toffoli
= X-1-n + Toffoli-(n-1)-(n-1)

ZeroRegTest (same for InvZeroRegTest)(with parameter k)
= 1-k-PL-X + (k-1)-SQ-Toffoli
= X-1-k + Toffoli-(k-1)-(k-1)

DIFFUSENODE
= 2-15-PL-H + 1-SQ-X + 2-SQ-ZeroNodeTest + 1-SQ-CNOT

[21]We follow the pseudocode and keep the ZERO operation.

DIFFUSEVI
= 2(n+r) H + 1-SQ-X + 2-SQ-ZeroNodeTest + 2-SQ-ZeroRegTest + 2-SQ-Toffoli + 1-SQ-CNOT

DIFFUSEGC
= 2(rbar+r) H + 1-SQ-X + 4-SQ-ZeroRegTest + 2-SQ-Toffoli + 1-SQ-CNOT

FetchT [22]
= Toffoli-R-nR + CompareIJ-2R-2R

FetchE
= Toffoli-R-R^2 + CompareIJ-2R-2R

StoreT[23]
= Toffoli-R-nR + CompareIJ-2R-2R

FetchStoreT
= ctrSwap-Rbar-Rbar + CompareIJrbar-2R-2R

FetchStoreE
= ctrSwap-3R-3R^2 + CompareIJ-2R-2R

ctrCTRincr
= OneRegTest-1-1 + CNOT-1-(2rbar - 3) + ZERO-1-(n+1) + CNOT-2-2
ctrCTRdecr
= ZeroNodeTest-1-1 + X-1-(2*rbar-3)+ CNOT-1-(2*rbar-3) + ZERO-1-(n+1) + X-2-2 + CNOT-2-2; **UPDATE**
= 2^r Swap
= CNOT-1536-1536

SWAP[24]
= $3n$ CNOT
= CNOT-3-45

QWSH
= 1-2n-PL-ZERO + 1-SQ-DIFFUSEVI + 2-SQ-FetchT + 2-SQ-FetchStoreE + 2-SQ-UPDATE + 2-SQ-StoreT + 1-n-PL-Swap
= CNOT(568600, 10257730) + H(182368, 3379400) + S (91182, 1689646) + T(273546, 5068938) + Tdag(364728, 6758584) + X(5, 49)

TestTriangleEdges
= TriangleTestT + TriangleEdgeSearch + TriangleTestTw
CNOT(1635981049840, 3051291585372) + H(527147551433, 901932671431) + S(263573650472, 450964825112) + T(790720951416, 1352894475336) + Tdag(1054294601888, 1803859300448) + X(25856362, 302180354)

[22]There are 3 types of FetchT (FetchT, FetchTr1, FetchTrbarr), the operations are the same but different in the value of the parameter (actually those parameters are close in value). Hence, we chose the largest one, FetchT, to represent.
[23]There are 2 types of StoreT (StoreT, StoreTrbarr), the operations are the same but different in the value of the parameter (actually those parameters are close in value). Hence, we chose the largest one, StoreT, to represent.
[24]Different from Swap, this SWAP operation acts on a quNode, not a qubit.

TriangleTestT[25]
$= \text{(Parellel)}((\sum_{i=0}^{R-1}\sum_{j=i+1}^{R-1}\sum_{k=j+1}^{R-1} 1)\times$ (4-SQ-Toffoli + 2-SQ-EvaluateOR + 1-SQ-CNOT + 1-SQ-ctrONE + 1-SQ-ZERO)
≈ [26] CNOT(52, 1156413440) + H (16, 355819520) + S (8, 177909760) + T (24, 533729280) + Tdag(32, 711639040) + X(4, 133432320)

TriangleTestTw[27]
$= \text{(Parallel)}(\sum_{i=0}^{R-1}\sum_{j=i+1}^{R-1} 1)\times$ (4-SQ-Toffoli + 2-SQ-EvaluateOR + 1-SQ-CNOT + 1-SQ-ctrONE + 1-SQ-ZERO)
= CNOT(52, 6802432) + H (16, 2093056) + S (8, 1046528) + T (24, 3139584) + Tdag(32, 4186112) + X(4, 784896)

TriangleEdgeSearch
= H-1-15 + t_g-SQ-GCQWalk + t_g-SQ-(2-SQ-ZeroRegTest (parameter CTR) + X-2-2 + 2-SQ-Toffoli + 1-SQ-CNOT) + t_g-SQ-DIFFUSENODE + t_g-SQ-InverseGCQWalk

≈ CNOT(1635981049736, 3050128369500) + H(527147551401, 901574758855) + S(263573650456, 450785868824) + T(790720951368, 1352357606472) + Tdag(1054294601824, 1803143475296) + X(25856354, 167963138)

GCQWalk
= 1-(Rbar×r + rbar + r)-PL-H + 1-(n + R + Rbar + 2rbar -1 + r + 1 + Rbar)-PL-ZERO + 2rbar-SQ-FetchT) + Rbar-SQ-FetchE + ((Rbar-1)Rbar/2)-SQ-(FetchT + 4-SQ-Toffoli + ctrCTRincr + FetchT) + Rbar- FetchE + $tbar_m$×(4-SQ-X + 2-SQ-ZeroRegTest (parameter CTR) + 2-SQ-Toffoli) + $tbar_m \times tbar_w$ GCQWStep

≈ CNOT(5760496497, 10739888468) + H(1856153297, 3174558943) + S(928076208, 1587274160) + T(2784228624, 4761822480) + Tdag(3712304832, 6349096640) + X(91040, 591392)

GCQWalkStep[28]
= 1-SQ-DIFFUSEGC + 2-SQ-FetchT + 1-SQ-FetchStoreT + 1-SQ-FetchE + Rbar-SQ-(FetchT + 4-SQ-Toffoli + ctrCTRdecr) + 1-SQ-CNOT + Rbar-SQ-FetchT + 1 Fetch T + 1 FetchE + 1 StoreT + r Swap + 1 FetchT + 1 FetchE + Rbar-SQ-FetchT + 1-SQ-CNOT + Rbar-SQ-(FetchT + 4-SQ-Toffoli + ctrCTRdecr + 1-SQ-FetchT) + 1-SQ-FetchE + 1-SQ-FetchStoreT + 2-SQ-FetchT

≈ CNOT(30544171, 56317483) + H(9841981, 16622961) + S(4920988, 8311452) + T(14762964, 24934356) + Tdag(19683952, 33245808) + X(517, 3359)

[25]We can do this operation in parallel since in our program we assign a new qubit for each iteration. So, the sum of the iteration will be multiplied to the occurrance, not the time slot.
[26] $\sum_{i=0}^{R-1}\sum_{j=i+1}^{R-1}\sum_{k=j+1}^{R-1} 1 = \sum_{t=1}^{R-1} t(R-1-t) = \frac{1}{6}(R-2)(R-1)R = 22238720$
[27]Same reason as TriangleTestT
[28]Those fetch and store related operations could have dependency.

```
(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Computation for parallel factors
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Vector Definition ; an 1 by 16 vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Data Structure V ={CNOT-time, CNOT-frequency,  ... (same for H, S, T, Tdag, X, Measure, Z)}


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear and then declare variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
ClearAll["Global`*"];


tm = 64;
tw = 22;
tbarm = 8;
tbarw = 22;
tg= 142;
n = 15;
r = 9;
rbar = 6;
Rbar = 64;
R=512;
ctr = 11;

RRR = 22238720;
RR = 130816;
RRbar = 2016;
tbarm = 8;
tbarw = 22;
tg=142;


(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define basic building blocks's values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
GToffoli = {6,6,2,2,1,1,3,3,4,4,0,0,0,0,0,0};
G01Toffoli = {0,6,0,2,0,1,0,3,0,4,0,0,0,0,0,0};
G10Toffoli = {6,0,2,0,1,0,3,0,4,0,0,0,0,0,0,0};
GZeroNodeTest = (n-1)*GToffoli + {0,0,0,0,0,0,0,0,0,0,1,n,0,0,0,0};
GZeroRegTestr = (r-1)*GToffoli + {0,0,0,0,0,0,0,0,0,0,1,r,0,0,0,0};
GZeroRegTestrbar = (rbar-1)*GToffoli + {0,0,0,0,0,0,0,0,0,0,1,rbar,0,0,0,0};
GZeroRegTestCTR = {0,0,0,0,0,0,0,0,0,0,1,ctr,0,0,0,0} + (ctr-1)*GToffoli;
G1CNOT = {1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
G01CNOT = {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
G10CNOT = {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
G01H =  {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0};
G10H =  {0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0};
G01X = {0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0};
G10X = {0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0};
G1X = {0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0};
G9M = {0,0,0,0,0,0,0,0,0,0,0,0,9,9,0,0};

GCompareIndexr = 2*r*G01CNOT + 2*G10CNOT;
(* be pessimistic by assuming all parameters are 15, even though it might be 9 sometimes *)
GOneRegTest = (n-1)*GToffoli + G1CNOT;
GCompareIJ = GCompareIndexr + GOneRegTest;
```

```
GCompareIndexrbar =  2*rbar*G01CNOT + 2*G10CNOT;
GCompareIJrbar = GCompareIndexrbar + GOneRegTest;

GZERO = 2*G1CNOT;
G10ZERO = 2*G10CNOT;
G01ZERO = 2*G01CNOT;
GctrSwap = 3*GToffoli;
GctrONE = 2*GToffoli + G1CNOT ;
GEvaluateOR = 1*GToffoli +{0,0,0,0,0,0,0,0,0,2,3,0,0,0,0};

GDIFFUSENODE = 2*GZeroNodeTest + 2*G10H + 2*n*G01H + G1CNOT + G1X;
GDIFFUSEVI = 2*GToffoli + 2*GZeroNodeTest + 2*GZeroRegTestr + 2*G10H + 2*(n+r)*G01H + G1CNOT+ G1X;
GDIFFUSEGC = 2*GToffoli +2*GZeroRegTestr+ 2*GZeroRegTestrbar+ 2*G10H + 2*(rbar+r)*G01H + G1CNOT+ G1X;

GFetchT = R*G10Toffoli + n*R*G01Toffoli +  2*R*GCompareIJ ;
GFetchE = R*G10Toffoli + R*R*G01Toffoli +  2*R*GCompareIJ ;
GStoreT = R*G10Toffoli + n*R*G01Toffoli +  2*R*GCompareIJ ;
GFetchStoreT = Rbar*GctrSwap +  2*Rbar*GCompareIJrbar ;
GFetchStoreE = 3*R*G10Toffoli + 3*R*R*G01Toffoli +  2*R*GCompareIJ ;
GctrCTRincr = GOneRegTest + (2*rbar-3)*G01CNOT + G10CNOT + G10ZERO + (n+1)*G01ZERO + 2*G1CNOT;


(* 2rbar - 1 less than n but greater than r, so we choose n to approximate *)
GctrCTRdecr = GZeroNodeTest + (2*rbar-3)*G01X + G10X + (2*rbar-3)*G01CNOT;
GctrCTRdecr =  GctrCTRdecr + G10CNOT+ G10ZERO + (n+1)*G01ZERO + 2G1X + 2*G1CNOT;
GtheUpdate = 3*R*G1CNOT;
GBigSwap = {3,3*n,0,0,0,0,0,0,0,0,0,0,0,0,0};
GQWSH = 2*G10H + 4*n*G01H + GDIFFUSEVI + 2*GFetchT + 2*GFetchStoreE + 2*GtheUpdate + 2*GStoreT + GBigSwap;


(*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute the factors for those larger modules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*)
GTriangleTestTSingleIteration = 4*GToffoli + 2*GEvaluateOR + G1CNOT + 1*GctrONE +  2*G1CNOT;
GTriangleTestT = GTriangleTestTSingleIteration;
GTriangleTestT[[2]] = RRR*GTriangleTestTSingleIteration[[2]];
GTriangleTestT[[4]] = RRR*GTriangleTestTSingleIteration[[4]];
GTriangleTestT[[6]] = RRR*GTriangleTestTSingleIteration[[6]];
GTriangleTestT[[8]] = RRR*GTriangleTestTSingleIteration[[8]];
GTriangleTestT[[10]] = RRR*GTriangleTestTSingleIteration[[10]];
GTriangleTestT[[12]] = RRR*GTriangleTestTSingleIteration[[12]];

GTriangleTestTw =  GTriangleTestTSingleIteration;
GTriangleTestTw[[2]] = RR*GTriangleTestTSingleIteration[[2]];
GTriangleTestTw[[4]] = RR*GTriangleTestTSingleIteration[[4]];
GTriangleTestTw[[6]] = RR*GTriangleTestTSingleIteration[[6]];
GTriangleTestTw[[8]] = RR*GTriangleTestTSingleIteration[[8]];
GTriangleTestTw[[10]] = RR*GTriangleTestTSingleIteration[[10]];
GTriangleTestTw[[12]] = RR*GTriangleTestTSingleIteration[[12]];

GGCQWalkStep = 1*GDIFFUSEGC+ 2*GFetchT+GFetchStoreT+GFetchE+Rbar(GFetchT+4*GToffoli + GctrCTRdecr);
GGCQWalkStep = GGCQWalkStep + G1CNOT + Rbar*(GFetchT) + GFetchT + GFetchE + GStoreT;
GGCQWalkStep = GGCQWalkStep + {0,0,3,3*r,0,0,0,0,0,0,0,0,0,0,0}+GFetchT + GFetchE + Rbar*GFetchT + G1CNOT;
GGCQWalkStep = GGCQWalkStep + Rbar*(GFetchT+4*GToffoli + GctrCTRdecr + GFetchT) + GFetchE + GFetchStoreT + 2*GFetchT;

GGCQWalk = {1,n+R+Rbar+ctr+r+1+Rbar, 1, Rbar*r + rbar + r,0,0,0,0,0,0,0,0,0,0,0}+2*rbar*GFetchT;
GGCQWalk = GGCQWalk + Rbar*GFetchE + RRbar*(GFetchT + 4*GToffoli + GctrCTRincr + GFetchT);
GGCQWalk = GGCQWalk + Rbar*GFetchE +tbarm*({0,0,0,0,0,0,0,0,0,4,4,0,0,0,0}+2*GZeroRegTestCTR +2*GToffoli);
GGCQWalk = GGCQWalk + tbarm*(tbarw*GGCQWalkStep);

GTriangleEdgeSearch = tg*(2*GZeroRegTestCTR + {0,0,0,0,0,0,0,0,0,2,2,0,0,0,0} + 4*GToffoli + G1CNOT + GDIFFUSENODE);
GTriangleEdgeSearch = GTriangleEdgeSearch + {0,0,1, n,0,0,0,0,0,0,0,0,0,0,0} + tg*2*GGCQWalk;
GTestTriangleEdges = GTriangleTestT + GTriangleEdgeSearch + GTriangleTestTw ;
```

```
(* Compute the parallel factor *)
GRI ={2, 16778, 1, 7708, 0,0,0,0,0,0,2,4612,0,0,0,0};
GSU = {4, 523264, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
GM = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 138511, 0, 0 };

PFCNOTA = (GRI[[2]] + GSU[[2]] + GTestTriangleEdges[[2]]*(2*tm+1) + GQWSH[[2]]*tm*tw );
PFCNOTB = (GRI[[1]] + GSU[[1]] + GTestTriangleEdges[[1]]*(2*tm+1) + GQWSH[[1]]*tm*tw );
PFCNOT = PFCNOTA/PFCNOTB;

PFHA = (GRI[[4]] + GSU[[4]] + GTestTriangleEdges[[4]]*(2*tm+1) + GQWSH[[4]]*tm*tw );
PFHB = (GRI[[3]] + GSU[[3]] + GTestTriangleEdges[[3]]*(2*tm+1) + GQWSH[[3]]*tm*tw );
PFH = PFHA/PFHB;

PFSA = (GRI[[6]] + GSU[[6]] + GTestTriangleEdges[[6]]*(2*tm+1) + GQWSH[[6]]*tm*tw );
PFSB = (GRI[[5]] + GSU[[5]] + GTestTriangleEdges[[5]]*(2*tm+1) + GQWSH[[5]]*tm*tw );
PFS = PFSA/PFSB;

PFTA = (GRI[[8]] + GSU[[8]] + GTestTriangleEdges[[8]]*(2*tm+1) + GQWSH[[8]]*tm*tw );
PFTB = (GRI[[7]] + GSU[[7]] + GTestTriangleEdges[[7]]*(2*tm+1) + GQWSH[[7]]*tm*tw );
PFT = PFTA/PFTB;

PFTDA = (GRI[[10]] + GSU[[10]] + GTestTriangleEdges[[10]]*(2*tm+1) + GQWSH[[10]]*tm*tw );
PFTDB = (GRI[[9]] + GSU[[9]] + GTestTriangleEdges[[9]]*(2*tm+1) + GQWSH[[9]]*tm*tw );
PFTD = PFTDA/PFTDB;

PFXA = (GRI[[12]] + GSU[[12]] + GTestTriangleEdges[[12]]*(2*tm+1) + GQWSH[[12]]*tm*tw );
PFXB = (GRI[[11]] + GSU[[11]] + GTestTriangleEdges[[11]]*(2*tm+1) + GQWSH[[11]]*tm*tw );
PFX = PFXA/PFXB;

(* Have to put GM in here since it is the only place where measurement takes place *)
PFMA = (GRI[[14]] + GSU[[14]] + GTestTriangleEdges[[14]]*(2*tm+1) + GQWSH[[14]]*tm*tw + GM[[14]]);
PFMB = (GRI[[13]] + GSU[[13]] + GTestTriangleEdges[[13]]*(2*tm+1) + GQWSH[[13]]*tm*tw + GM[[13]]);
PFM = PFMA/PFMB;
```