QMDD Implementation

Nicolas Guillemot

Abstract

An implementation of the Quantum Multiple-Valued Decision Diagram (QMDD) data structure was built, which supports building QMDDs from circuits specified in the TFC format. This implementation supports a wide variety of quantum gates, accurately computes edge weights for the subset of supported quantum gates, can implement gates in terms of other gates, and is general to p-valued logic. The program can output a visualization of the QMDD as a GraphViz dot file, which is currently its main mode of operation. Future work would include supporting conjugate and adjoint operators, and implementing matrix/vector multiplication to support simulating the circuit for a given input state.

Introduction

For this project, an implementation of the Quantum Multiple-Valued Decision Diagram (QMDD) [1] was created based on the descriptions found in several works [1] [2] [3]. The purpose of this report is to document the software design of this QMDD implementation. During the development of the software, effort was put into writing code that gives a clean exposition to the algorithms used in the implementation of the QMDD, so hopefully it can also be used as a reference for others who want to study the algorithm.

Base knowledge of QMDDs is assumed of the reader, and uninitiated readers are recommended to read the referred works. In short, QMDDs are an optimized storage method for the matrices used in quantum and multiple-valued logic. QMDDs implement a recursive subdivision of quadrants of the matrix, where sub-matrices are represented as a directed acyclic graph data structure, which allows identical sub-matrices to be deduplicated by sharing the same node in the graph. This deduplication is especially effective in the simulation of quantum circuits, due to the recurring structured patterns in the matrices used in such simulations.

This report begins with a description of the user interface for the program. From there, the high-level architecture of the program is described, including a description of its major components. We'll then dive into the details of the implementation of the algorithms used to construct a QMDD from a quantum circuit. Finally, opportunities for future work will be enumerated in the conclusion.

The code is around 2.5 KLOC of standard C++ code, and has no dependencies other than calling GraphViz dot from the command line to generate an image from the graph description. The code for the implementation is available on GitHub under the MIT license: https://github.com/nlguillemot/qmdd

User Interface

The input to the QMDD builder is a program in the "tfc" format, as described on Dmitri Maslov's "Reversible Logic Synthesis Benchmarks Page" [4]. This format allows the specification of a quantum circuit as a list of gates. The filename of the program specification is passed as a command line argument to the QMDD builder, and the QMDD builder will parse, decode, and display the resulting QMDD.

The tfc format described on Maslov's page only specifies the description of Toffoli gates and Fredkin gates, but doesn't specify the syntax for many useful quantum gates. To address this, the language was extended to handle a wide variety of quantum gates. Namely, the following gates are supported.

Name	TFC opcode	Matrix
X aka. Toffoli aka. NOT	t	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Fredkin aka. $SWAP$	f	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Y aka. Pauli Y	y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Z aka. Pauli Z	z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
V aka. $\sqrt{NOT}$	v	$\begin{pmatrix} \frac{1+i}{2} & \frac{1-i}{2} \\ \frac{1-i}{2} & \frac{1+i}{2} \end{pmatrix}$
V^+ aka. $\left(\sqrt{NOT}\right)^{-1}$	v'	$\begin{pmatrix} \frac{1-i}{2} & \frac{1+i}{2} \\ \frac{1+i}{2} & \frac{1-i}{2} \end{pmatrix}$
H aka. Hadamard	h	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
T aka. T-gate	q ("quarter of π")	$\begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix}$
T^+ aka. T-gate^{-1}	q'	$\begin{pmatrix} 1 & 0 \\ 0 & e^{-\frac{i\pi}{4}} \end{pmatrix}$
S aka. Phase	s	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
S^+ aka. Phase^{-1}	s'	$\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$

Figure 1: Gate types handled by the extended tfc format.

Each gate type can optionally be conditionally executed based on a set of control variables. If no control variables are given, the gate is unconditionally executed.

Some gates were left unimplemented. Notably, the $e^{\frac{i\pi}{8}}$ phase shift gate and the $\sqrt{SWAP}$ gate. The $e^{\frac{i\pi}{8}}$ gate was unimplemented because it requires an irrational number other than $\sqrt{2}$, though it shouldn't be difficult to extend the system to handle another irrational term. The $\sqrt{SWAP}$ gate was left unimplemented because I couldn't find a way to construct it from single-target gates (unlike $SWAP$).

As an example, this program implements $CNOT$. The output of the QMDD builder is shown on the right.

```
.v a,b
.i a,b
.o a,b
BEGIN
t2 a,b
END
```

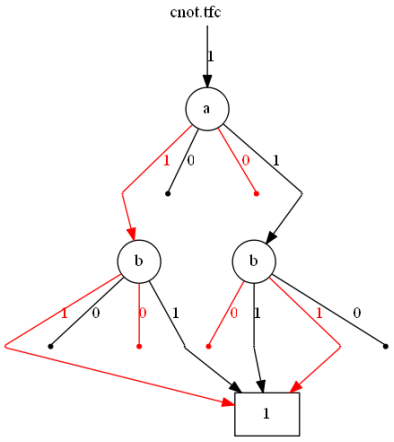
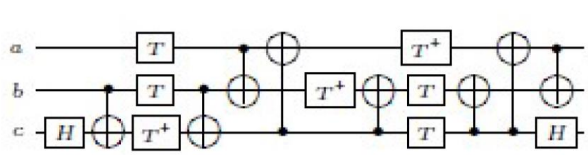


Figure 2: Basic tfc program implementing a CNOT gate.

The program begins by declaring its variables, inputs, and outputs. The body is between the "BEGIN" and "END" terminals, where each statement is one gate of the circuit. The name of the gate begins the statement, like how "t" means "Toffoli" above. The number of inputs to the gate follows its name. The target is the last variable of the statement, and all other variables are controls. Fredkin gate is an exception, which has 2 target variables (the two variables that it swaps).

As a more complicated example, consider the following program, which implements a 2-control Toffoli gate using H gates and T gates. A visualization of the circuit (from [5]) is shown in the middle to help understand it. The output of the program is shown on the right, which is indeed a 2-control Toffoli.

```
.v a,b,c
.i a,b,c
.o a,b,c
BEGIN
h1 c
t2 b,c
q1 a
q1 b
q'1 c
t2 b,c
t2 a,b
t2 c,a
q'1 b
t2 c,b
q'1 a
q1 b
q1 c
t2 c,b
t2 c,a
t2 a,b
h1 c
END
```



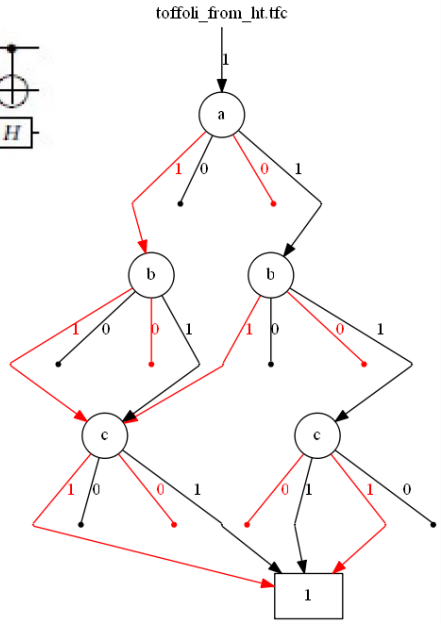


Figure 3: tfc specification of a 2-control Toffoli circuit using H gates and T gates.

Overall Architecture

In this section, the overall architecture of the program is described, including its three main phases: parsing, decoding, and dot graph generation. The phases of the program can be seen directly in the program's main function, as shown below.

```
program_spec spec;
try {
    spec = parse(spec_str.c_str());
}
catch (const std::exception& e) {
    throw std::runtime_error(infilename + ":" + e.what());
}

qmdd::edge root;
qmdd dd = qmdd(spec.num_variables);
decode(spec, dd, &root);

std::string outfilename = infilename + ".dot";
write_dot(infilename.c_str(), spec, dd, root, outfilename.c_str());
display_dot(outfilename.c_str());
```

Figure 4: the QMDD builder's main function

Parsing

The parsing stage, in the call to `parse()`, reads the input tfc file and converts it to a program specification. The program specification contains some metadata, like the number of variables, and the names of the variables. It also contains a list of quantum gates, represented as a stream of instructions, using a variable-length encoding. Each instruction is encoded as a list of `int`s corresponding to an opcode, a parameter count, and a list of parameters (equal in length to the previously specified number of parameters.)

The parser itself is implemented with a finite state machine and a recursive descent parser. The state machine keeps track of state, like whether the parser is reading the tfc header or the circuit's body, and the recursive descent parser is used to read each statement in the tfc file. The parser detects and reports syntactical and semantic errors by throwing an exception with a line number and error message.

Decoding

The decoding stage, in the call to `decode()`, is the most complex stage of this program, and its inner workings will be described in more detail later in this document. At a high level, the decoder reads one instruction at a time from the program specification, constructs a QMDD that represents the gate that implements the instruction, and appends that gate to the circuit being built. The decoder also returns the root of the QMDD generated by decoding the circuit, which allows further processing on this QMDD.

Dot Graph Output

Once the circuit has been decoded, the resulting subgraph of QMDD nodes is outputted to a GraphViz[1] dot file, which allows visualizing the QMDD. This is rendered by traversing the root's subgraph of the QMDD, while outputting its nodes and edges. Some tricky graph layout constraints had to be used to make the output look like the diagrams used to represent QMDDs in the literature.

[1] http://www.graphviz.org/

QMDD Builder Design

In this section, explanations of key components of the QMDD builder are given. Namely, the algorithm used to construct gates, the computation of edge weights, gate microcode, and multiple-valued logic.

QMDD Gate Construction

The decoder converts each gate into an equivalent matrix, and multiplies that matrix with the matrix that represents the circuit so far. The matrix for a single-target gate is built from bottom to top. Each circuit line is merged with the gate's matrix using matrix additions and Kronecker products. Control lines are handled differently if the line is above or below the target. There is a more detailed explanation in the referred works, so I'll avoid repeating It here, but I'll contribute pseudo-code for this routine.

`if_true` and `if_false` correspond to the 2x2 matrices `[0,0;0,1]` and `[1,0;0,0]`, respectively. `identity` is the 2x2 identity matrix `[1,0;0,1]`. `identities[var]` refers to a QMDD made from Kronecker products of identity matrices, including the specified variable and every other variable below it. `gate_matrix` is the 2x2 matrix that represents the current gate, like `[0,1;1,0]` for a NOT gate. Gate operations are implemented similarly to Bryant's classic if-then-else `apply()` function [6].

```
root = edge{w=1, node=terminal}
foreach gate in circuit, do
  active = edge{w=1, node=terminal}
  inactive = edge{w=0, node=terminal}
  foreach var from bottom line to top line, do
    if var is above target, do
      if var is control line, do
        active = edge{w=1, node{var, if_true}} ⊗ active;
        inactive = (edge{w=1, node{var, if_false}} ⊗ identities[next var])
                 + (edge{w=1, node{var, if_true}} ⊗ inactive);
      else, do
        active = edge{w=1, node{var, identity}} ⊗ active;
        inactive = edge{w=1, node{var, identity}} ⊗ inactive;
      end
    else if var is target, do
      active = (edge{w=1, node{var, identity}} ⊗ inactive)
             + (edge{w=1, node{var, gate_matrix}} ⊗ active);
    else if var is below target, do
      if var is control line, do
        active = (edge{w=1, node{var, if_false}} ⊗ identities[next var])
               + (edge{w=1, node{var, if_true}} ⊗ active);
      else, do
        active = edge{w=1, node{var, identity}} ⊗ active;
      end
    end
  end
  root = active * root;
end
```

Figure 5: Pseudo-code to construct a QMDD from a series of gates in a circuit.

Edge Weights

Each edge in the QMDD has a weight, and these weights are complex numbers. If we restrict ourselves to the gates listed in the beginning of this document, we need to handle real and imaginary integers, rational numbers, as well as irrational numbers. A general quantum circuit simulation package might decide to represent these edge weights using a wide floating point number type, like "long double" in C. However, this makes it harder to get exact answers to our calculations, and numerical errors make it more difficult to determine that two mathematically identical submatrices are indeed identical.

To counter this problem, a specialized numerical type was defined. A simple rational number class would make it possible to handle most of the matrices considered in this project, but it wouldn't be enough to handle the matrices that include a factor of $\sqrt{2}$. Conveniently, since $\sqrt{2}$ is the only irrational number in these equations (including also the $e^{\frac{i\pi}{4}}$ terms), we can treat it as a special case. Thus, the following representation was used:

$$\left(a + b\sqrt{2}\right) + i\left(c + d\sqrt{2}\right)$$

$$a, b, c, d \in \mathbb{Q}$$

Using this representation, each edge weight is represented by 4 rational numbers, each of which is made of two integers (a numerator and a denominator.) With this representation, it becomes possible to implement addition, subtraction, multiplication, and division, without any loss of precision (assuming the rational type does not overflow.) The $\sqrt{2}$ term is algebraically treated in a similar way to i, since i is irreducible but $i * i = -1$. Similarly, $\sqrt{2}$ is irreducible, and $\sqrt{2} * \sqrt{2} = 2$. With an existing rational number package, defining and implementing the arithmetic operations on this type requires doing only a bit of simple algebra.

With this implementation of the weights, the QMDD builder could convert long sequences of complex irrational gates into very simple gates, as was demonstrated earlier in this document. However, this system cannot handle arbitrary rotations, and can't handle other irrational numbers, like those that come from the $e^{\frac{i\pi}{8}}$ phase shift gate.

Gate Microcode

The gate construction algorithm described earlier only works for single-target gates, and would have to be modified to implement multiple-target gates, such as the Fredkin $SWAP$ gate. Fortunately, the Fredkin gate can be implemented in terms of Toffoli gates. Therefore, the Fredkin gate is implemented in the QMDD decoder by simply emitting instructions corresponding to the equivalent sequence of Toffoli gates.

There may be a limitation to this approach, since I couldn't easily find a way to implement the $\sqrt{SWAP}$ gate (which has two targets) in terms of other gates. Maybe it's just that a bit more research is necessary.

Multiple-Valued Logic

This QMDD builder was written in a way that is general to p-valued logic, meaning it can support circuits that are binary, ternary, etc. Gates used just need to be defined in terms of p. Currently, only a few basic gates are defined for $p > 2$, so this is an area of potential improvement.

Conclusions & Future Work

The QMDD builder implemented can parse and decode quantum circuits specified with an extended form of the TFC language. This report focused mainly on the algorithm used to construct gates, the special number format used to accurately do computations on edge weights, the implementation of gates in terms of other gates, and the handling of multiple-valued logic.

The QMDD builder does have a few other features that weren't discussed in this report, like the use of computed tables for nodes and weights. However, this QMDD builder lacks many features from the literature. For example, it lacks garbage collection (opting instead to have a fixed upper bound). The conjugate and adjoint operations on QMDDs are also not implemented. Finally, there is no method to simulate the circuit given an initial state, which requires multiplying the QMDD's matrix by a vector containing the initial state. None of these features are impossible or especially difficult to implement, but time just ran out. Thus, this program is mainly useful as a visualization of QMDDs through its GraphViz dot output.

References

[1] D. M. Miller and M. A. Thornton, "QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits," in *Proceedings of the 36th International Symposium on Multiple-Valued Logic*, 2006.

[2] P. Niemann, R. Wille, D. M. Miller, M. A. Thornton and R. Drechsler, "QMDDs: Efficient Quantum Function Representation and Manipulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 35, no. 1, January 2016.

[3] D. M. Miller and M. A. Thornton, Multiple Valued Logic: Concepts and Representations, Morgan & Claypool, 2008.

[4] D. Maslov, "Reversible Logic Synthesis Benchmarks Page (Main\Machine-Readable Format Description)," [Online]. Available: http://webhome.cs.uvic.ca/~dmaslov/mach-read.html. [Accessed 2017].

[5] M. Miller, *CSC 485B-552 Lecture Notes,* 2017.

[6] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* vol. 35, no. 8, pp. 677-691, 1986.