MRF Compiler Reference Manual

Shyamal Suhana Chandra

November 17, 2025

Contents

1 Introduction

This reference manual provides complete documentation for the MRF Compiler, a comprehensive tool for converting probabilistic graphical models into Markov Random Fields (MRF) and subsequently into quantum circuit representations. The compiler supports multiple quantum computing frameworks and provides a flexible command-line interface for easy integration into existing workflows.

1.1 Purpose

The MRF Compiler bridges the gap between classical probabilistic modeling and quantum computing, enabling researchers and practitioners to:

- Convert graphical models to quantum circuits automatically
- Export to multiple quantum computing frameworks
- Leverage quantum algorithms for probabilistic inference
- Integrate quantum computing into existing machine learning pipelines

1.2 Scope

This manual covers:

- Installation and setup procedures
- Command-line interface and options
- Input file format specifications
- Output format details for each framework
- API reference for programmatic usage
- Framework-specific integration guides
- Examples and use cases
- Troubleshooting and common issues

2 Installation

2.1 System Requirements

- **Operating System**: Linux, macOS, or Windows (with WSL or MinGW)
- **Compiler**: C++17 compatible compiler (GCC 7+, Clang 5+, or MSVC 2017+)
- **Build System**: Make
- **Memory**: Minimum 512MB RAM (2GB recommended for large graphs)
- **Disk Space**: 50MB for installation

2.2 Building from Source

2.2.1 Linux/macOS

```
# Clone the repository
git clone https://github.com/shyamalchandra/MRFcompiler.git
cd MRFcompiler

# Build the compiler
make

# Optional: Install system-wide
sudo make install
```

2.2.2 Windows

```
# Using MinGW or MSYS2
git clone https://github.com/shyamalchandra/MRFcompiler.git
cd MRFcompiler
mingw32-make

# Or using Visual Studio
# Open the project in Visual Studio and build
```

2.3 Verification

After building, verify the installation:

```
./mrf_compiler --help
```

You should see the help message with available options.

3 Command-Line Interface

3.1 Basic Usage

The basic command syntax is:

./mrf_compiler [options] [input_file] [output_file]

3.2 Command-Line Options

Option	Description
-f, --framework <name>	Specify output framework. Supported values: qasm, qiskit, cirq, pennylane, qsharp, braket, qulacs, tfq. Default: qasm
-a, --all	Export to all supported frameworks. Creates multiple output files with framework-specific extensions.

Option	Description
-h, --help	Display help message with usage information and available options.
-v, --verbose	Enable verbose output, showing detailed conversion steps and debugging information.
-o, --output <file>	Specify output file path. If not provided, output is written to stdout or a default filename based on input.

3.3 Examples

3.3.1 Basic Usage

```
# Convert to OpenQASM (default)
./mrf_compiler example.txt output.qasm

# Convert to Qiskit
./mrf_compiler -f qiskit example.txt circuit.py

# Convert to Cirq
./mrf_compiler -f cirq example.txt circuit.py
```

3.3.2 Export to All Frameworks

```
# Generate output for all frameworks
./mrf_compiler -a example.txt

# This creates:
# - output_qasm.qasm
# - output_qiskit.py
# - output_cirq.py
# - output_pennylane.py
# - output_qsharp.qs
# - output_braket.py
# - output_qulacs.py
# - output_tfq.py
```

3.3.3 Verbose Mode

```
# Get detailed conversion information
./mrf_compiler -v -f qiskit example.txt circuit.py
```

4 Input Format

4.1 File Structure

The input file is a plain text file with a simple line-based format. Each line specifies either the graph type, a node, or an edge.

4.2 Graph Type Declaration

The first line must specify the graph type:

```
TYPE directed
```

or

```
TYPE undirected
```

4.3 Node Declaration

Nodes are declared with the following syntax:

```
NODE <id> <name> [num_states]
```

- `<id>`: Unique integer identifier for the node (starting from 0)
- `<name>`: Alphanumeric name for the node
- `[num`$_{s}tates] : Optional number of states (default : 2 for binary)$

4.4 Edge Declaration

Edges are declared with the following syntax:

```
EDGE <from> <to> [directed]
```

- `<from>`: Source node ID
- `<to>`: Target node ID
- `[directed]`: Optional flag (only relevant for undirected graphs, ignored for directed)

4.5 Complete Example

```
TYPE undirected
NODE 0 A 2
NODE 1 B 2
NODE 2 C 2
NODE 3 D 2
EDGE 0 1
EDGE 1 2
EDGE 2 3
EDGE 0 3
```

This creates a 4-node undirected graph forming a square (cycle).

4.6 Directed Graph Example

```
TYPE directed
NODE 0 A 2
NODE 1 B 2
NODE 2 C 2
EDGE 0 1
EDGE 0 2
EDGE 1 2
```

This creates a directed graph where A is a parent of both B and C, and B is a parent of C.

5 Output Formats

5.1 OpenQASM 2.0

OpenQASM is the standard quantum assembly language. The output is a text file that can be used with any OpenQASM-compatible quantum simulator or hardware.

File Extension: `.qasm`

Example Output:

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
creg c[3];

h q[0];
h q[1];
h q[2];
cx q[0], q[1];
rz(0.5) q[1];
cx q[0], q[1];
```

5.2 Qiskit

Qiskit is IBM's quantum computing framework. The output is a Python file with a function that returns a `QuantumCircuit` object.

File Extension: `.py`

Example Output:

```python
from qiskit import QuantumCircuit

def create_circuit():
    qc = QuantumCircuit(3, 3)
    qc.h(0)
    qc.h(1)
    qc.h(2)
    qc.cx(0, 1)
```

```python
    qc.rz(0.5, 1)
    qc.cx(0, 1)
    return qc

if __name__ == "__main__":
    circuit = create_circuit()
    circuit.draw(output='mpl')
```

5.3 Cirq

Cirq is Google's quantum computing framework. The output is a Python file with a function that returns a `cirq.Circuit` object.

File Extension: .py

Example Output:

```python
import cirq

def create_circuit():
    qubits = [cirq.GridQubit(0, i) for i in range(3)]
    circuit = cirq.Circuit()
    circuit.append([cirq.H(qubits[i]) for i in range(3)])
    circuit.append(cirq.CNOT(qubits[0], qubits[1]))
    circuit.append(cirq.rz(0.5)(qubits[1]))
    circuit.append(cirq.CNOT(qubits[0], qubits[1]))
    return circuit
```

5.4 PennyLane

PennyLane is Xanadu's quantum machine learning library. The output is a Python file with a QNode-decorated function.

File Extension: .py

Example Output:

```python
import pennylane as qml

dev = qml.device('default.qubit', wires=3)

@qml.qnode(dev)
def circuit():
    qml.Hadamard(wires=0)
    qml.Hadamard(wires=1)
    qml.Hadamard(wires=2)
    qml.CNOT(wires=[0, 1])
    qml.RZ(0.5, wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.state()
```

5.5 Q#

Q# is Microsoft's quantum programming language. The output is a Q# source file with an operation.

File Extension: .qs

Example Output:

```
namespace MRFCircuit {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Math;

    operation CreateCircuit() : Unit {
        using (qubits = Qubit[3]) {
            H(qubits[0]);
            H(qubits[1]);
            H(qubits[2]);
            CNOT(qubits[0], qubits[1]);
            Rz(0.5, qubits[1]);
            CNOT(qubits[0], qubits[1]);
        }
    }
}
```

5.6 AWS Braket

AWS Braket is Amazon's quantum computing service. The output is a Python file with a function that returns a `braket.Circuit` object.

File Extension: .py

Example Output:

```python
from braket.circuits import Circuit

def create_circuit():
    circuit = Circuit()
    circuit.h(0)
    circuit.h(1)
    circuit.h(2)
    circuit.cnot(0, 1)
    circuit.rz(1, 0.5)
    circuit.cnot(0, 1)
    return circuit
```

5.7 Qulacs

Qulacs is a fast quantum circuit simulator. The output is a Python file with a function that returns a `qulacs.QuantumCircuit` object.

File Extension: .py

Example Output:

```python
from qulacs import QuantumCircuit

```

```python
def create_circuit():
    circuit = QuantumCircuit(3)
    circuit.add_H_gate(0)
    circuit.add_H_gate(1)
    circuit.add_H_gate(2)
    circuit.add_CNOT_gate(0, 1)
    circuit.add_RZ_gate(1, 0.5)
    circuit.add_CNOT_gate(0, 1)
    return circuit
```

5.8 TensorFlow Quantum

TensorFlow Quantum is Google's quantum machine learning framework. The output is a Python file that creates a tensor representation.

File Extension: .py

Example Output:

```python
import tensorflow_quantum as tfq
import cirq

def create_circuit():
    qubits = [cirq.GridQubit(0, i) for i in range(3)]
    circuit = cirq.Circuit()
    circuit.append([cirq.H(qubits[i]) for i in range(3)])
    circuit.append(cirq.CNOT(qubits[0], qubits[1]))
    circuit.append(cirq.rz(0.5)(qubits[1]))
    circuit.append(cirq.CNOT(qubits[0], qubits[1]))
    return tfq.convert_to_tensor([circuit])
```

6 API Reference

6.1 Graph Module

6.1.1 GraphicalModel Class

The `GraphicalModel` class represents a probabilistic graphical model.

Methods:

- `void addNode(int id, const std::string& name, int num_states = 2)`: Add a node to the graph
- `void addEdge(int from, int to, bool directed = true)`: Add an edge between nodes
- `Node* getNode(int id)`: Get a node by ID
- `Edge* getEdge(int from, int to)`: Get an edge between two nodes
- `std::vector<int> getNeighbors(int node_id) const`: Get all neighbors of a node

- `bool hasEdge(int from, int to) const`: Check if an edge exists
- `void print() const`: Print the graph structure

6.2 MRF Module

6.2.1 MRF Class

The `MRF` class represents a Markov Random Field.

Methods:

- `void addNode(int id, const std::string& name, int num_states = 2)`: Add a node
- `void addClique(const std::vector<int>& nodes)`: Add a clique
- `void setCliquePotential(int clique_idx, const std::vector<double>& potential)`: Set clique potential
- `void print() const`: Print the MRF structure
- `int getTotalStates() const`: Get total number of states

6.2.2 Conversion Functions

- `MRF convertToMRF(const GraphicalModel& gm)`: Convert graphical model to MRF
- `void moralizeGraph(GraphicalModel& gm)`: Moralize a directed graph
- `std::vector<Clique> findMaximalCliques(const GraphicalModel& gm)`: Find all maximal cliques

7 Framework Integration

7.1 Qiskit Integration

After generating a Qiskit circuit:

```python
from output_qiskit import create_circuit
from qiskit import Aer, execute

circuit = create_circuit()
backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
```

7.2 Cirq Integration

After generating a Cirq circuit:

```python
from output_cirq import create_circuit
import cirq

circuit = create_circuit()
simulator = cirq.Simulator()
result = simulator.simulate(circuit)
print(result)
```

7.3 PennyLane Integration

After generating a PennyLane circuit:

```python
from output_pennylane import circuit
import pennylane as qml

result = circuit()
print(result)
```

8 Examples

8.1 Example 1: Simple Chain Graph

Input (`chain.txt`):

```
TYPE undirected
NODE 0 A 2
NODE 1 B 2
NODE 2 C 2
EDGE 0 1
EDGE 1 2
```

Command:

```
./mrf_compiler -f qiskit chain.txt chain_circuit.py
```

8.2 Example 2: Directed Graph with Moralization

Input (`directed.txt`):

```
TYPE directed
NODE 0 A 2
NODE 1 B 2
NODE 2 C 2
EDGE 0 1
EDGE 0 2
```

This creates a v-structure that requires moralization (adding edge between B and C).

Command:

```
./mrf_compiler -f cirq directed.txt directed_circuit.py
```

8.3 Example 3: Export to All Frameworks

Command:

```
./mrf_compiler -a example.txt
```

This generates output files for all 8 supported frameworks.

9 Troubleshooting

9.1 Common Issues

9.1.1 Compilation Errors

Problem: Build fails with compilation errors.

Solution: Ensure you have a C++17 compatible compiler. Update your compiler or use a newer version.

9.1.2 File Not Found

Problem: `./mrf_compiler: No such file or directory`

Solution: Ensure you're in the correct directory and the build completed successfully. Run `make` again.

9.1.3 Invalid Input Format

Problem: Error parsing input file.

Solution: Check that your input file follows the correct format:

- First line must be `TYPE directed` or `TYPE undirected`
- Node IDs must be unique integers starting from 0
- Edge node IDs must reference existing nodes

9.1.4 Framework Import Errors

Problem: Generated Python files fail to import framework modules.

Solution: Install the required framework:

```
pip install qiskit         # For Qiskit
pip install cirq          # For Cirq
pip install pennylane     # For PennyLane
pip install amazon-braket-sdk  # For AWS Braket
pip install qulacs         # For Qulacs
pip install tensorflow-quantum  # For TensorFlow Quantum
```

9.1.5 Large Graph Performance

Problem: Compilation is slow for large graphs.
Solution:

- Clique finding is exponential in worst case
- Consider using smaller graphs or approximate methods
- Increase available memory
- Use verbose mode to identify bottlenecks

10 Advanced Usage

10.1 Custom Potential Functions

While the current version uses default potential functions, future versions will support custom potential specifications in the input file.

10.2 Circuit Optimization

The generated circuits are basic representations. For production use, consider:

- Gate optimization using framework-specific tools
- Circuit compilation for specific hardware
- Noise-aware compilation for NISQ devices

10.3 Performance Considerations

- **Graph Size**: Performance degrades with graph size, especially for dense graphs
- **Clique Count**: Graphs with many maximal cliques take longer to process
- **Output Format**: Some frameworks generate more verbose output than others

11 Best Practices

1. **Start Small**: Test with small graphs before scaling up
2. **Verify Output**: Always check generated circuits for correctness
3. **Use Version Control**: Keep input files and generated outputs in version control
4. **Document Models**: Add comments to input files describing the model structure
5. **Test Multiple Frameworks**: Compare outputs across different frameworks
6. **Optimize Circuits**: Use framework-specific optimization tools after generation

12 Contributing

The MRF Compiler is an open-source project. Contributions are welcome! Areas for contribution include:

- Additional framework support
- Performance optimizations
- Extended input format support
- Documentation improvements
- Bug fixes and testing

13 License

14 References

- Qiskit Documentation: `https://qiskit.org/documentation/`
- Cirq Documentation: `https://quantumai.google/cirq`
- PennyLane Documentation: `https://pennylane.ai/`
- Q# Documentation: `https://docs.microsoft.com/azure/quantum/`
- AWS Braket Documentation: `https://docs.aws.amazon.com/braket/`