tweedledum Documentation

Release alpha-v1

Bruno Schmitt

Dec 17, 2018

Contents

CHAPTER 1

Installation

tweedledum is a header-only C++-17 library. Just add the include directory of tweeldedum to your include directories, and you can integrate it into your source files using

```
#include <tweedledum/tweedledum.hpp>
```

1.1 Alpha Disclaimer

tweedledum is in version Alpha. Hence, the software is still under active development and not feature complete, meaning the API is subject to big changes. This is released for developers or users who are comfortable living on the absolute bleeding edge.

1.2 Requirements

We tested building tweedledum on Mac OS and Linux using:

- Clang 6.0.0
- Clang 7.0.0
- GCC 7.3.0
- GCC 8.1.0.

If you experience that the system compiler does not suffice the requirements, you can manually pass a compiler to CMake using:

```
cmake -DCMAKE_CXX_COMPILER=/path/to/c++-compiler ..
```

1.3 Building the examples

The included *CMake build script* can be used to build the tweedledum library examples on a wide range of platforms. CMake is freely available for download from http://www.cmake.org/download/.

CMake works by generating native makefiles or project files that can be used in the compiler environment of your choice. The typical workflow starts with:

```
mkdir build      # Create a directory to hold the build output.
cd build
```

To build the *examples* set the `TWEEDLEDUM_EXAMPLES` CMake variable to `TRUE`:

```
cmake -DTWEEDLEDUM_EXAMPLES=TRUE <path/to/tweedledum>
```

where *<path/to/tweedledum>* is a path to the `tweedledum` repository.

If you are on a *nix system, you should now see a Makefile in the current directory. Now you can build the library by running **make**.

All `*.cpp` files in the *examples/* directory will be compiled to its own executable which will have the same name. For example, the file `examples/hello_world.cpp` will generate the executable `hello_world`.

Once the examples have been built you can invoke **./examples/<name>** to run it:

```
./examples/hello_world
```

1.4 Building the documentation

To build the documentation you need the following software installed on your system:

- Python with pip and virtualenv
- Doxygen

First generate makefiles or project files using CMake as described in the previous section. Then compile the `doc` target/project, for example:

```
make doc
```

This will generate the HTML documentation in `doc/html`.

CHAPTER 2

Tutorial

Todo: Finish writing

CHAPTER 3

Change Log

3.1 Alpha-v1.0

- Initial network and gate interfaces:
- Gate implementations:
- Network implementations:
- Algorithms:
- I/O
- Utility data structures:

CHAPTER 4

Acknowledgments

The tweedledum library is maintained by Bruno Schmitt with contributions from Mathias Soeken and Fereshte Mozafari. Let me know if your contribution is not listed or mentioned incorrectly and I'll make it right.

CHAPTER 5

References

CHAPTER 6

The tweedledum philosophy

CHAPTER 7

The Standard Gate Set

Below is a summary of the key gates used in tweedledum

Name(s)	Symbol	tweedledum symbol	Matrix
Identity	I	gate_set::identity	(1
0			
0 1			
Hadamard	H	gate_set::hadamard	$\frac{1}{\sqrt{2}}$ (1
1			
1 -1			
Arbitrary rotations			
X Rotation	Rx	gate_set::rotation_x	$(\cos\frac{\theta}{2}$
-isin $\frac{\theta}{2}$			
-isin $\frac{\theta}{2}$ $\cos\frac{\theta}{2}$			
Y Rotation	Ry	gate_set::rotation_y	
Z Rotation	Rz	gate_set::rotation_z	$(e^{-i\theta}$
0			
0 $e^{i\theta}$			
Named Rotations			
Pauli X, NOT	X	gate_set::pauli_x	(0
1			
1 0			
T	T	gate_set::t	(1
0			
0 $e^{i\frac{\pi}{4}}$			
T dagger	T†	gate_set::t_dagger	
Phase	S	gate_set::phase	(1
0			
0 i			
Phase dagger	S†	gate_set::phase_dagger	
Pauli Z, Phase flip	Z	gate_set::pauli_z	(1
0			
0 -1			
Controlled gates			
Control NOT	CNOT	gate_set::cx	(1
0 0	0		
0 1	0	0	
0 0	0	1	
0 0	1	0	
Control Z	CZ	gate_set::cz	(1
0 0	0		
0 1	0	0	
0 0	1	0	
0 0	0	-1	
Multiple Control NOT, Toffoli		gate_set::mcx	
Multiple Control Z		gate_set::mcz	

CHAPTER 8

Gate interface API

A `Gate` is an `gate_base` that is applied to a collection of qubits. Those qubits are identified by a `qid` given by a `network`.

This page describes the interface of a quantum gate data structure in *tweedledum*.

> **Warning:** This part of the documentation makes use of a class called `gate`. This class has been created solely for the purpose of creating this documentation and is not meant to be used in code.

8.1 Mandatory types and constants

A gate must expose the following compile-time constants:

```
static constexpr uint32_t max_num_qubits;
static constexpr uint32_t network_max_num_qubits;
```

The struct `is_gate_type` can be used to check at compile time whether a given type contains all required types and constants to implement a network type. It should be used in the beginning of an algorithm that expects a gate type:

```
template<typename Gate>
class network {
        static_assert(is_gate_type_v<Gate>, "Gate is not a gate type");
};
```

8.2 Methods

8.2.1 Constructors

`class gate`

Public Functions

gate (*gate_base* **const** &*op*, qubit_id *target*)
Construct a single qubit gate.

Parameters

- `op`: the operation (must be a single qubit operation).
- `target`: qubit identifier of the target.

gate (*gate_base* **const** &*controlled_op*, qubit_id *control*, qubit_id *target*)
Construct a controlled gate.

Parameters

- `controlled_op`: the operation (must be a two qubit controlled operation).
- `control`: qubit identifier of the control.
- `target`: qubit identifier of the target.

gate (*gate_base* **const** &*unitary_op*, std::vector<qubit_id> **const** &*controls*, std::vector<qubit_id> **const** &*targets*)
Construct a gate using vectors.

Parameters

- `unitary_op`: the operation (must be unitary operation).
- `control`: qubit(s) identifier of the control(s).
- `targets`: qubit identifier of the target.

8.2.2 Properties

class gate

Public Functions

uint32_t **num_controls** () **const**
Return the number of controls.

uint32_t **num_targets** () **const**
Returns the number of targets.

8.2.3 Iterators

class gate

Public Functions

template <**typename** Fn>
void **foreach_control**(Fn &&*fn*) **const**

Calls `fn` on every target qubit of the gate.

The paramater `fn` is any callable that must have one of the following two signatures.

- `void(qubit_id)`
- `bool(qubit_id)`

If `fn` returns a `bool`, then it can interrupt the iteration by returning `false`.

template <**typename** Fn>
void **foreach_target**(Fn &&*fn*) **const**

Calls `fn` on every target qubit of the gate.

The paramater `fn` is any callable that must have one of the following signature.

- `void(qubit_id)`

CHAPTER 9

Network interface API

This page describes the interface of a quantum network data structure in *tweedledum*.

> **Warning:** This part of the documentation makes use of a class called `network`. This class has been created solely for the purpose of creating this documentation and is not meant to be used in code. Custom network implementation do **not** have to derive from this class, but only need to ensure that, if they implement a function of the interface, it is implemented using the same signature.

9.1 Mandatory types and constants

The interaction with a network data structure is performed using four types for which no application details are assumed. The following four types must be defined within the network data structure. They can be implemented as nested type, but may also be exposed as type alias.

template <typename G>
class network

Public Types

template<>
using base_type = network

Type referring to itself.

The `base_type` is the network type itself. It is required, because views may extend networks, and this type provides a way to determine the underlying network type.

template<>
using gate_type = G

Type representing a gate.

A `Gate` is an operation that can be applied to a collection of qubits. It could be a meta operation, such as, primary input and a primary output, or a unitary operation gate.

struct node_type
Type representing a node.

A `node` is a node in the network. Each node must contains a gate.

struct storage_type
Type representing the storage.

A `storage` is some container that can contain all data necessary to store the network. It can constructed outside of the network and passed as a reference to the constructor. It may be shared among several networks. A `std::shared_ptr<T>` is a convenient data structure to hold a storage.

Further, a network must expose the following compile-time constants:

```
static constexpr uint32_t min_fanin_size;
static constexpr uint32_t max_fanin_size;
```

The struct `is_network_type` can be used to check at compile time whether a given type contains all required types and constants to implement a network type. It should be used in the beginning of an algorithm that expects a network type:

```
template<typename Network>
void algorithm(Network const& ntk) {
        static_assert(is_network_type_v<Network>, "Network is not a network type");
}
```

9.2 Methods

9.2.1 Constructors

template <typename G>
class network

9.2.2 Qubits and Ancillae

template <typename G>
class network

Public Functions

auto **add_qubit**(std::string **const** &*qlabel*)
Creates a labeled qubit in the network and returns its `qid`

auto **add_qubit**()
Creates a unlabeled qubit in the network and returns its `qid`

Since all qubits in a network must be labeled, this function will create a generic label with the form: qN, where N is the `qid`.

9.2.3 Structural properties

template <**typename** G>
class network

Public Functions

uint32_t **size**() **const**
Returns the number of nodes.

uint32_t **num_qubits**() **const**
Returns the number of qubits.

uint32_t **num_gates**() **const**
Returns the number of gates, i.e., nodes that hold unitary operations.

9.2.4 Node iterators

template <**typename** G>
class network

Public Functions

template <**typename** Fn>
void **foreach_cqubit**(Fn &&*fn*) **const**
Calls `fn` on every qubit in the network.

The paramater `fn` is any callable that must have one of the following three signatures.

- `void(uint32_t qid)`
- `void(string const& qlabel)`
- `void(uint32_t qid, string const& qlabel)`

template <**typename** Fn>
void **foreach_cinput**(Fn &&*fn*) **const**
Calls `fn` on every input node in the network.

The paramater `fn` is any callable that must have one of the following two signatures.

- `void(node_type const& node)`
- `void(node_type const& node, uint32_t node_index)`

template <**typename** Fn>
void **foreach_coutput**(Fn &&*fn*)
Calls `fn` on every output node in the network.

The paramater `fn` is any callable that must have one of the following two signatures.

- `void(node_type const& node)`
- `void(node_type const& node, uint32_t node_index)`

template <**typename** Fn>

void **foreach_cgate**(Fn &&*fn*) **const**

Calls `fn` on every unitrary gate node in the network.

The paramater `fn` is any callable that must have one of the following four signatures.

- `void(node_type const& node)`
- `void(node_type const& node, uint32_t node_index)`
- `bool(node_type const& node)`
- `bool(node_type const& node, uint32_t node_index)`

If `fn` returns a `bool`, then it can interrupt the iteration by returning `false`.

template <**typename** Fn>
void **foreach_cnode**(Fn &&*fn*) **const**

Calls `fn` on every node in the network.

The paramater `fn` is any callable that must have one of the following four signatures.

- `void(node_type const& node)`
- `void(node_type const& node, uint32_t node_index)`
- `bool(node_type const& node)`
- `bool(node_type const& node, uint32_t node_index)`

If `fn` returns a `bool`, then it can interrupt the iteration by returning `false`.

CHAPTER 10

Implementations

10.1 Gate base

A custom gate implementation **must** derive from the `gate_base` class.

`class gate_base`

Simple class to hold information about the operation of a gate.

Subclassed by tweedledum::mcmt_gate, tweedledum::mcst_gate

10.2 Custom gates

All gate implementations are located in *tweedledum/gates/*:

Interface method	mcst	mcmt
	Constants	
`max_num_qubits`	3	32
`network_max_num_qubits`		32
	Properties	
`num_controls`	✓	✓
`num_targets`	✓	✓
	Iterators	
`foreach_control`	✓	✓
`foreach_target`	✓	✓

10.3 Networks

All network implementations are located in *tweedledum/networks/*:

Interface method	netlist
	I/O and ancillae qubits
`add_qubit`	✓
`add_ancilla`	
	Structural properties
`size`	✓
`num_qubits`	✓
`num_gates`	✓
	Iterators
`foreach_cqubit`	✓
`foreach_cinput`	✓
`foreach_coutput`	✓
`foreach_cgate`	✓
`foreach_cnode`	✓

CHAPTER 11

Decomposition

Decomposition: is the process of breaking down in parts or elements.

High-level quantum algorithms are technology-independent, that is, allow arbitrary quantum gates, and do not take architectural constraints into account. Quite often, these algorithms involve quantum gates acting on *n* qubits. In order to execute such an algorithm in a quantum computer it is necessary to decompose these gates in an series of simpler gates.

The *tweedledum* library implements several decomposition algorithms. The following table lists all decomposition algorithms that are currently provided in *tweedledum*

11.1 Barenco decomposition

Header: `tweedledum/algorithms/decomposition/barenco.hpp`

11.1.1 Parameters

`struct barenco_params`

Parameters for `barenco_decomposition`.

11.1.2 Algorithm

`template <typename` Network>
Network `tweedledum::`**`barenco_decomposition`**(Network **`const`** &*src*, *barenco_params params* = {})

Barenco decomposition.

Decomposes all Multiple-controlled Toffoli gates with more than `controls_threshold` controls into Toffoli gates with at most `controls_threshold` controls. This may introduce one additional helper qubit called ancilla.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `num_controls`

Required network functions:

- `add_gate`
- `foreach_cqubit`
- `foreach_cgate`
- `rewire`
- `rewire_map`

11.2 Direct Toffoli (DT) decomposition

Header: `tweedledum/algorithms/decomposition/dt.hpp`

11.2.1 Algorithm

template <**typename** Network>
Network `tweedledum::`**`dt_decomposition`**(Network **const** &*src*)

Direct Toffoli (DT) decomposition.

Decomposes all Multiple-controlled Toffoli gates with 2, 3 or 4 controls into Clifford+T. Also decompose all Multiple-controlled Z gates with 2 controls into Clifford+T. This may introduce one additional helper qubit called ancilla.

These Clifford+T represetations were obtained using techniques inspired by *[Mas16]* and given in *[AMMR13]*

Required gate functions:

- `foreach_control`
- `foreach_target`
- `num_controls`

Required network functions:

- `add_gate`
- `foreach_cqubit`
- `foreach_cgate`
- `rewire`
- `rewire_map`

CHAPTER 12

Synthesis

The *tweedledum* library implements several synthesis algorithms. These take as input a function in terms of some representation and return a reversible or quantum circuit. The following table lists all synthesis algorithms that are currently provided in *tweedledum*.

12.1 CNOT-Patel Synthesis for linear reversible functions

Header: `tweedledum/algorithms/synthesis/cnot_patel.hpp`

12.1.1 Parameters

`struct cnot_patel_params`
Parameters for `cnot_patel`.

Public Members

bool **`allow_rewiring`** = false
Allow rewiring.

bool **`best_partition_size`** = false
Search for the best parition size.

uint32_t **`partition_size`** = 1u
Partition size.

12.1.2 Algorithm

`template` <**`class`** Network, **`class`** Matrix>

Network `tweedledum::cnot_patel` (Matrix **const** &*matrix*, *cnot_patel_params* *params* = { })

CNOT Patel synthesis for linear circuits.

This algorithm is based on the work in *[PMH08]*.

The following code shows how to apply the algorithm to the example in the original paper.

```
std::vector<uint32_t> rows = {0b000011,
                                0b011001,
                         0b010010,
                         0b111111,
                         0b111011,
                         0b011100};
bit_matrix_rm matrix(6, rows);
cnot_patel_params parameters;
parameters.allow_rewiring = false;
parameters.best_partition_size = false;
parameters.partition_size = 2u;
auto network = cnot_patel<netlist<mcst_gate>>(matrix, parameters);
```

Parameters

- `matrix`: The square matrix representing a linear reversible circuit.
- `params`: The parameters that configure the synthesis process. See *cnot_patel_params* for details.

Warning: doxygenfunction: Unable to resolve multiple matches for function "tweedledum::cnot_patel" with arguments (Network&, std::vector<uint32_t> const&, Matrix const&, cnot_patel_params) in doxygen xml output for project "tweedledum" from directory: doxyxml/xml. Potential matches:

```
- template <class Network, class Matrix>
  Network tweedledum::cnot_patel(Matrix const&, cnot_patel_params)
- template <class Network, class Matrix>
  void tweedledum::cnot_patel(Network&, std::vector<qubit_id> const&, Matrix const&,
↪ cnot_patel_params)
```

12.2 Decomposition-based synthesis (DBS)

Header: `tweedledum/algorithms/synthesis/dbs.hpp`

This synthesis algorithm is based on the following property of reversible functions: Any reversible function $f : \mathbb{B}^n \to \mathbb{B}^n$ can be decomposed into three reversible functions $f_r \circ f' \circ f_l$, where f_l and f_r are single-target gates acting on target line x_i and f' is a reversible function that does not change in x_i.

12.2.1 Parameters

struct dbs_params

Parameters for `dbs`.

Public Members

bool **verbose** = false
Be verbose.

12.2.2 Algorithm

template <**class** Network, **class** STGSynthesisFn>
Network tweedledum::**dbs**(std::vector<uint32_t> *perm*, STGSynthesisFn &&*stg_synth*, *dbs_params params* = {})
Reversible synthesis based on functional decomposition.

This algorithm implements the decomposition-based synthesis algorithm proposed in *[DVVR08]*. A permutation is specified as a vector of 2^n different integers ranging from 0 to $2^n - 1$.

```
std::vector<uint32_t> permutation{{0, 2, 3, 5, 7, 1, 4, 6}};
auto network = dbs<netlist<mcst_gate>>(permutation, stg_from_spectrum());
```

Parameters

- `perm`: A permutation
- `stg_synth`: Synthesis function for single-target gates
- `params`: Parameters (see *dbs_params*)

12.3 Gray synthesis for {CNOT, Rz} circuits

Header: `tweedledum/algorithms/synthesis/gray_synth.hpp`

12.3.1 Parameters

struct gray_synth_params
Parameters for `gray_synth`.

12.3.2 Algorithm

Warning: doxygenfunction: Unable to resolve multiple matches for function “tweedledum::gray_synth” with arguments (Network&, std::vector<uint32_t> const&, parity_terms const&, gray_synth_params) in doxygen xml output for project “tweedledum” from directory: doxyxml/xml. Potential matches:

```
- template <class Network>
  Network tweedledum::gray_synth(uint32_t, parity_terms const&, gray_synth_params)
- template <class Network>
  void tweedledum::gray_synth(Network&, std::vector<qubit_id> const&, parity_terms␣
↪const&, gray_synth_params)
```

template <**class** Network>

Network tweedledum::**gray_synth**(uint32_t *num_qubits*, parity_terms **const** &*parities*, *gray_synth_params params* = {})

Gray synthesis for {CNOT, Rz} networks.

This algorithm is based on the work in *[AAM17]*.

The following code shows how to apply the algorithm to the example in the original paper.

```

```

Return {CNOT, Rz} network

Parameters

- `num_qubits`: Number of qubits
- `parities`: List of parities and rotation angles to synthesize
- `params`: The parameters that configure the synthesis process. See *gray_synth_params* for details.

12.4 Linear synthesis for {CNOT, Rz} circuits

Header: `tweedledum/algorithms/synthesis/linear_synth.hpp`

12.4.1 Parameters

struct linear_synth_params

Parameters for `linear_synth`.

12.4.2 Algorithm

Warning: doxygenfunction: Unable to resolve multiple matches for function "tweedledum::linear_synth" with arguments (Network&, std::vector<uint32_t> const&, parity_terms const&, linear_synth_params) in doxygen xml output for project "tweedledum" from directory: doxyxml/xml. Potential matches:

```
- template <class Network>
  Network tweedledum::linear_synth(uint32_t, parity_terms const&, linear_synth_
↪params)
- template <class Network>
  void tweedledum::linear_synth(Network&, std::vector<qubit_id> const&, parity_
↪terms const&, linear_synth_params)
```

template <class Network>
Network tweedledum::**linear_synth**(uint32_t *num_qubits*, parity_terms **const** &*parities*, *linear_synth_params params* = {})

Linear synthesis for small {CNOT, Rz} networks.

Synthesize all linear combinations.

Return {CNOT, Rz} network

Parameters

- `num_qubits`: Number of qubits
- `parities`: List of parities and rotation angles to synthesize
- `params`: The parameters that configure the synthesis process. See *linear_synth_params* for details.

Function	Description	Expects	Returns
cnot_patel	CNOT Patel synthesis for linear circuits.	Linear matrix	{CNOT} network
dbs	Reversible synthesis based on functional decomposition.	Permutation	Quantum or reversible circuit
gray_synth	Gray synthesis for {CNOT, Rz} networks.	List of parities and rotation angles to synthesize	{CNOT, Rz} network
linear_synth	Linear synthesis for small {CNOT, Rz} networks.	List of parities and rotation angles to synthesize	{CNOT, Rz} network

CHAPTER 13

Open QASM 2.0

template <typename Network>
void tweedledum::**write_qasm**(Network **const** &*network*, std::string **const** &*filename*)

Writes network in OPENQASM 2.0 format into a file.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `op`

Required network functions:

- `num_qubits`
- `foreach_cnode`

Parameters

- `network`: A quantum network
- `filename`: Filename

template <typename Network>
void tweedledum::**write_qasm**(Network **const** &*network*, std::ostream &*os*)

Writes network in OPENQASM 2.0 format into output stream.

An overloaded variant exists that writes the network into a file.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `op`

Required network functions:

- `foreach_cnode`
- `num_qubits`

Parameters

- `network`: A quantum network
- `os`: Output stream

CHAPTER 14

Quil

template <**typename** Network>
void `tweedledum::`**`write_quil`**(Network **const** &*network*, std::string **const** &*filename*)

Writes network in quil format into a file.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `op`

Required network functions:

- `foreach_cnode`
- `foreach_cqubit`
- `num_qubits`

Parameters

- `network`: A quantum network
- `filename`: Filename

template <**typename** Network>
void `tweedledum::`**`write_quil`**(Network **const** &*network*, std::ostream &*os*)

Writes network in quil format into output stream.

An overloaded variant exists that writes the network into a file.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `op`

Required network functions:

- `foreach_cnode`
- `foreach_cqubit`
- `num_qubits`

Parameters

- `network`: A quantum network
- `os`: Output stream

CHAPTER 15

Write to qpic file format

template <**typename** Network>
void tweedledum::**write_qpic**(Network **const** &*network*, std::string **const** &*filename*, bool *color_marked_gates* = false)

Writes network in qpic format into a file.

Required gate functions:

- `foreach_control`
- `foreach_target`
- `op`

Required network functions:

- `foreach_cnode`
- `foreach_cqubit`
- `num_qubits`

Parameters

- `network`: A quantum network
- `filename`: Filename
- `color_marked_gates`: Flag to draw marked nodes in red

template <**typename** Network>
void tweedledum::**write_qpic**(Network **const** &*network*, std::ostream &*os*, bool *color_marked_gates* = false)

Writes network in qpic format into output stream.

An overloaded variant exists that writes the network into a file.

Required gate functions:

- `foreach_control`

- `foreach_target`
- `op`

Required network functions:

- `foreach_cnode`
- `foreach_cqubit`
- `num_qubits`

Parameters

- `network`: A quantum network
- `os`: Output stream
- `color_marked_gates`: Flag to draw marked nodes in red

CHAPTER 16

Write to unicode string

template <**typename** Network>
void tweedledum::**write_unicode**(Network **const** &*network*, std::string **const** &*filename*)

Writes a network in Unicode format format into a file.

Required gate functions:

- `op`
- `foreach_control`
- `foreach_target`

Required network functions:

- `foreach_cgate`
- `num_qubits`

Parameters

- `network`: A quantum network
- `filename`: Filename

template <**typename** Network>
void tweedledum::**write_unicode**(Network **const** &*network*, std::ostream &*os* = std::cout)

Writes a network in Unicode format format into a output stream.

Required gate functions:

- `op`
- `foreach_control`
- `foreach_target`

Required network functions:

- `foreach_cgate`

- num_qubits

Parameters

- network: A quantum network
- os: Output stream (default: std::cout)

CHAPTER 17

Angle

`class angle`

Simple class to represent rotation angles.

A angle can be defined symbolically or numerically. The numeric value of a rotation angle is given in radians (rad).

CHAPTER 18

Indices and tables

- genindex
- modindex
- search

Bibliography

[AAM17] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the CNOT-complexity of CNOT-phase circuits. *arXiv preprint arXiv:1712.01859*, 2017. URL: https://arxiv.org/abs/1712.01859.

[AMMR13] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013. URL: https://ieeexplore.ieee.org/abstract/document/6516700.

[DVVR08] Alexis De Vos and Yvan Van Rentergem. Young subgroups for reversible computers. *Advances in Mathematics of Communications*, 2(2):183–200, 2008. URL: http://dx.doi.org/10.3934/amc.2008.2.183, doi:10.3934/amc.2008.2.183.

[Mas16] Dmitri Maslov. Advantages of using relative-phase toffoli gates with an application to multiple control toffoli optimization. *Physical Review A*, 93(2):022311, 2016. URL: https://journals.aps.org/pra/abstract/10.1103/PhysRevA.93.022311.

[PMH08] Ketan N. Patel, Igor L. Markov, and John P. Hayes. Optimal synthesis of linear reversible circuits. *Quantum Information & Computation*, 8(3):282–294, 2008. URL: http://www.rintonpress.com/xxqic8/qic-8-34/0282-0294.pdf.

Index

T