A Tensor Network based Decision Diagram for Representation of Quantum Circuits

Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying

Abstract—Tensor networks have been successfully applied in simulation of quantum physical systems for decades. Recently, they have also been employed in classical simulation of quantum computing, in particular, random quantum circuits. This paper proposes a decision diagram style data structure, called TDD (Tensor Decision Diagram), for more principled and convenient applications of tensor networks. This new data structure provides a compact and canonical representation for quantum circuits. By exploiting circuit partition, the TDD of a quantum circuit can be computed efficiently. Furthermore, we show that the operations of tensor networks essential in their applications (e.g., addition and contraction), can also be implemented efficiently in TDDs. A proof-of-concept implementation of TDDs is presented and its efficiency is evaluated on a set of benchmark quantum circuits. It is expected that TDDs will play an important role in various automation tasks related to quantum circuits, including but not limited to equivalence checking, error detection, synthesis, simulation, and verification.

I. INTRODUCTION

Google’s recent demonstration of quantum supremacy on its 53-qubit quantum processor Sycamore [1] has confirmed that quantum computers can indeed complete tasks much more efficiently than the most advanced traditional computers. Quantum devices of similar sizes have also been developed at IBM, Intel, IonQ, and Honeywell. It is widely believed that quantum processors with several hundreds of qubits will very likely appear in the next 5-10 years. The rapid growth of the size of quantum computing hardware motivates people to develop effective techniques for synthesis, optimisation, testing and verification of quantum circuits.

Mathematically, quantum circuits can be represented as unitary matrices, which transform initial quantum states (represented as vectors) to desired output states. The size of this matrix representation grows exponentially with the size of the quantum system, which makes it a great challenge to even simulate a quantum random circuit with a modest size and a shallow depth. Existing matrix-based packages like Qiskit (https://qiskit.org/) and the Google TensorNetwork [25] are very efficient, store such a matrix as a complete array, whose size quickly exceeds the memory limit. For example, it requires 64GB memory to store the functionality of a 16-qubit quantum circuit if each matrix entry is represented in data type complex128.

In order to alleviate the challenge and to provide a compact, canonical, and efficient representation for quantum functionalities, several decision diagram style data structures have been proposed, including Quantum Information Decision Diagrams (QIADs) [26] and Quantum Multiple-Valued Decision Diagrams (QMDDs) [23]. QMDD is a variant of Algebraic Decision Diagrams (ADDs) [2] by restricting values to complex numbers, which are indexed by integers, and interleaving row and column variables in the variable ordering. In contrast, QMDD partitions a transformation matrix into four submatrices of equal size, which in turn are partitioned similarly, and uses shared nodes to represent submatrices differing in only a constant coefficient. Evaluations in [23] showed that QMDDs offer a compact representation for large unitary (transformation) matrices. Consequently, they provide a compact and canonical representation for the functionality of quantum circuits. Indeed, QMDDs have been successfully used in simulation [29] and equivalence checking [10, 9] of quantum circuits as well as verifying the correctness of quantum circuits compilation [27].

Tensor networks provide a flexible way to represent quantum circuits and have been successfully employed in the classical simulation of quantum computing in the last few years. By observing that quantum circuits are a special class of tensor networks, Pednault et al. [24] exploited the flexibility of tensor computations with circuit partition and tensor slicing methods, and broke the 49-qubit barrier of that time in the simulation of quantum circuits. Later on, the size and depth of quantum circuits which can be simulated employing tensor network and the simulation time have been significantly improved (see, e.g., [5, 17, 11, 12, 16]). Tensor networks can also be applied in computing the functionality of a quantum circuit. Indeed, it can be computed in essentially any order, which in turn greatly affects the calculation efficiency. For a quantum circuit with low tree-width, by exploiting an optimal contraction order, the tensor representation of the quantum circuit can be computed in time polynomial in the size of the circuit [18]. While it is in general NP-hard to find an optimal contraction order, one may exploit heuristics like circuit partition [24], tree decomposition [18], and hyper-optimisation approaches [13], which have been demonstrated as very useful for simulating quantum circuits.

Inspired by the success of tensor networks in the classical simulation of quantum circuits, this paper aims to introduce a novel decision diagram, called Tensor Decision Diagram...
(TDD for short), for tensor networks. As a new data structure, TDD can further explore the flexibility of tensor networks in a more principled way, while overcoming the serious memory bottleneck of matrix-based representations.

While it is observed that the Boole-Shannon expansion commonly used in the design of decision diagrams is “not a basic decomposition for quantum mechanical phenomena [25]”, tensors, as multidimensional linear maps with complex values, do enjoy Boole-Shannon style expansions. This observation lays the foundation of our design of TDD.

TDDs have several important features that warrant their applicability. Analogous to reduced ordered binary decision diagrams (ROBDD) for Boolean functions [7], redundant nodes or nodes representing the same tensor in a TDD can be removed or merged so that shared nodes are used as much as possible. The canonicity result (Theorem [3]) guarantees that, up to variable ordering, each quantum circuit has a unique reduced TDD representation. An efficient algorithm (Alg. [1]) is also designed to generate the reduced TDD representation of a quantum functionality (e.g., a quantum gate or a part of a quantum circuit). Moreover, we show that basic TDD operations such as addition and contraction can be implemented efficiently. As QMDD, TDD provides a universal, compact and canonical representation for quantum circuits, which is vital in various design automation tasks.

In the remainder of this paper, after a brief review of quantum circuits and QMDD in Sec. [I] and of tensor networks in Sec. [III], we introduce our new data structure TDD in Sec. [IV]. The construction and implementation of basic tensor operations are presented in Sec. [V]. After that, we show how to compute the TDD representation of a quantum circuit in a circuit partition way in Sec. [VI]. Experimental results are reported and analysed in Sec. [VII]. The last section concludes the paper and briefly discusses several topics for future research. Most technical proofs as well as detailed experimental results are presented in the appendix.

II. BACKGROUND

For convenience of the reader, we review some basic concepts about quantum circuits and the Quantum Multi-value Decision Diagram (QMDD) in this section.

A. Quantum Circuits

The most basic concept in quantum computing is the qubit, which is the counterpart of bit in classical computing. The state of a qubit is often represented in Dirac notation

\[ |\varphi\rangle := \alpha_0 |0\rangle + \alpha_1 |1\rangle, \]

where \(\alpha_0\) and \(\alpha_1\) are complex numbers, called the amplitudes of \(|\varphi\rangle\), and satisfy \(|\alpha_0|^2 + |\alpha_1|^2 = 1\). We also use the vector \([\alpha_0, \alpha_1]^T\) to represent a single-qubit state. In general, an \(n\)-qubit quantum state can be represented as a \(2^n\)-dimensional complex vector \([\alpha_0, \alpha_1, \ldots, \alpha_{2^n-1}]^T\).

The evolution of a quantum system is described by a unitary transformation. In quantum computing, it is usually called a quantum gate. A quantum gate has a unique unitary matrix representation in a predefined orthonormal basis. Fig. [I] shows several such examples. The state after applying a specific transformation can be obtained by multiplying the corresponding unitary matrix and the vector that represents the input quantum state. For example, the output state resulting from applying a Hadamard gate to an input state \([\alpha_0, \alpha_1]^T\) is calculated as follows

\[ \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \alpha_0 + \alpha_1 \\ \alpha_0 - \alpha_1 \end{bmatrix}. \]

More generally, an \(n\)-qubit quantum gate is represented as a \(2^n \times 2^n\)-dimensional unitary transformation matrix.

A quantum circuit consists of a set of qubits and a sequence of elementary quantum gates. Given an input state to the qubits involved, the quantum gates in a quantum circuit will be applied to the input state in a sequential manner. The functionality of an \(n\)-qubit quantum circuit can also be described by a \(2^n \times 2^n\)-dimensional unitary transformation matrix.

B. Quantum Multi-value Decision Diagram

Quantum Multi-value Decision Diagram (QMDD) [19] is a decision diagram based data structure which provides a compact and canonical representation for quantum states and transformation matrices.

The main idea of QMDD is to recursively partition a \(2^n \times 2^n\) transformation matrix into submatrices till matrix elements are reached. The QMDD of \(M\) is constructed as follows: First, we introduce a root node, representing the original matrix. The root node has four successors, denoting the submatrices obtained by partitioning \(M\) into four with the same size. Each child node is then further expanded in the same manner. Suppose, in some step, a node corresponding to a matrix element is obtained. Then this node is regarded as a terminal node labelled 1, while its corresponding matrix element will be assigned as the weight of its incoming edge. The obtained decision diagram may have redundant nodes and weight-0 edges. After proper normalisation and reduction, we have a reduced decision diagram representation of \(M\), which is unique up to the order of variables.

Example 1. Shown in Fig. [2] is the QMDD representation of the controlled-T gate, where the node labeled with \(y_0\) represents the original matrix representation of the controlled-T gate and the two 0 attached to it represent the upper right and bottom left sub-matrices which are all 0-matrices. The two nodes labeled with \(y_1\) represent the upper left and bottom right sub-matrices.
which are, respectively, the identity matrix and the matrix of the T gate.

III. TENSOR AND TENSOR NETWORK

Before describing our data structure TDD, let us briefly recall the basic idea and notations of tensor networks.

A. Basic concepts

A tensor is a multidimensional linear map associated with a set of indices. In this paper, we assume that each index takes value in \( \{0, 1\} \). That is, a tensor with index set \( I = \{x_1, \ldots, x_n\} \) is simply a mapping \( \phi : \{0, 1\}^I \rightarrow \mathbb{C} \), where \( \mathbb{C} \) is the field of complex numbers. Sometimes, to emphasise the index set, we denote such a tensor by \( \phi_{x_1, \ldots, x_n} \) or \( \phi_{\vec{x}} \), and its value on the evaluation \( \{x_i \mapsto a_i\}, 1 \leq i \leq n \) by \( \phi_{x_1, \ldots, x_n}(a_1, \ldots, a_n) \), or simply \( \phi_{\vec{a}} \) or even \( \phi(\vec{a}) \) when there is no confusion. The number \( n \) of the indices of a tensor is called its rank. Scalars, 2-dimensional vectors, and 2 \( \times \) 2 matrices are rank 0, rank 1, and rank 2 tensors, respectively.

The most important operation between tensors is contraction. The contraction of two tensors is a tensor obtained by summing up over shared indices. Specifically, let \( \gamma_{\vec{x}, \vec{z}} \) and \( \xi_{\vec{y}, \vec{z}} \) be two tensors which share a common index set \( \vec{z} \). Then their contraction is a new tensor \( \phi_{\vec{x}, \vec{y}} \) with

\[
\phi_{\vec{x}, \vec{y}}(\vec{a}, \vec{b}) = \sum_{\vec{c} \in \{0, 1\}^I} \gamma_{\vec{x}, \vec{z}}(\vec{a}, \vec{c}) \cdot \xi_{\vec{y}, \vec{z}}(\vec{b}, \vec{c}).
\]

Another useful tensor operation is slicing, which corresponds to the cofactor operation of Boolean functions. Let \( \phi \) be a tensor with index set \( I = \{x, x_1, \ldots, x_n\} \). The slicing of \( \phi \) with respect to \( x = c \) with \( c \in \{0, 1\} \) is a tensor \( \phi_{|x=c} \) over \( I' = \{x_1, \ldots, x_n\} \) given by

\[
\phi_{|x=c}(\vec{a}) := \phi(c, \vec{a})
\]

for any \( \vec{a} \in \{0, 1\}^n \). We call \( \phi_{|x=0} \) and \( \phi_{|x=1} \) the negative and positive slicing of \( \phi \) with respect to \( x \), respectively. We say an index \( x \in I \) is essential for \( \phi \) if \( \phi_{|x=0} \neq \phi_{|x=1} \).

A tensor network is an undirected graph \( G = (V, E) \) with zero or multiple open edges, where each vertex \( v \in V \) represents a tensor and each edge a common index associated with the two adjacent tensors. By contracting connected tensors (i.e., vertices in \( V \)), with an arbitrary order, we get a rank \( m \) tensor, where \( m \) is the number of open edges of \( G \). This tensor, which is independent of the contraction order, is also called the tensor representation of the tensor network. Interested readers are referred to [18] and [4] for more detailed introduction.

B. Quantum circuits as tensor networks

The quantum state of a qubit \( x \) with vector representation \([\alpha_0, \alpha_1] \) can be described as a rank 1 tensor \( \phi_x \), where \( \phi_x(0) = \alpha_0 \) and \( \phi_x(1) = \alpha_1 \). Moreover, a single-qubit gate with input qubit \( x \) and output qubit \( y \) can be represented as a rank 2 tensor \( \phi_{xy} \). Note that for tensor representation, we do not distinguish between input and output indices, information about which can be naturally implied when tensors are interpreted as gates or circuits. For example, the tensor representation of a Z-gate, with \( x \) the input and \( y \) the output qubit, is \( \phi_{xy}(00) = 1, \phi_{xy}(01) = \phi_{xy}(10) = 0, \phi_{xy}(11) = -1 \). Likewise, an \( n \)-qubit gate is represented as a rank \( 2^n \) tensor.

A little thought shows that a quantum circuit is naturally a tensor network if we view gates as tensors as above. In such a tensor network, each vertex (tensor) represents a quantum state or a quantum gate and each edge a common index of two adjacent tensors. The functionality of any quantum circuit involving \( n \) qubits is naturally represented as a tensor of rank \( 2^n \), by contracting all the tensors involved, instead of a \( 2^n \times 2^n \) transformation matrix. This shift of perspective not only decreases our cognitive load, potentially, it will also provide a more concise representation of quantum functionality.

Example 2. Consider the circuit shown in Fig. 3. Regarding each gate as a tensor (cf. Fig. 1), Fig. 4 shows the tensor network representation of the circuit. By contracting the tensor network, we obtain the tensor representation of the circuit

\[
\phi_{z_2 x_1 z_3 y_3} = \sum_{x_0, a_1, a_2, b_2 = 0} T(a_0 a_1) H(b_0 b_1) C X(a_1 b_1 a_2 b_2) T(a_2 a_3) H(b_2 b_3).
\]

It is straightforward to check that this tensor indeed gives the functionality of the circuit presented in Fig. 3. For example, \( \phi_{z_0 x_1 z_0 y_3}(1111) = -i \) corresponds to the fact that the circuit maps \( |11\rangle \) to \(-i |11\rangle \).
Given a tensor $\phi_x$ and $x_i, x_j \in \mathbb{Z}$, if $\phi_x(a) = 0$ whenever $a_i \neq a_j$, we slightly abuse the notation to use an identical index for both $x_i$ and $x_j$. For example, the tensor for $Z$ gate can be written as $\phi_{xx}$ with $\phi_{xx}(0) = 1$ and $\phi_{xx}(1) = -1$. Similarly, CX gate can be represented as a tensor $\phi_{xx,y0,y2}$ with $\phi_{xx,y0,y2}(abc) = a \cdot (b \oplus c) + \pi \cdot b \oplus c$, where $\pi$, for example, is the complement of $a$. In \cite{24}, edges formed by identical indices are called hyper-edges.

**Example 3.** For the tensor network shown in Fig. 4, the four indices $x_0, x_1, x_2, x_3$ can all be represented by the same index $x_0$ since the two $T$ gates are diagonal and the $CX$ gate is block diagonal. Thus, the tensor network can be modified as the graph shown in Fig. 5, where the dotted line is a hyper-edge and the corresponding tensor becomes $\phi_{x_0,x_0,x0,x0}^4$.

**IV. Tensor Decision Diagram**

To fully exploit the benefit of tensor network representation of quantum circuits and the circuit partition technique, a suitable data structure — Tensor Decision Diagram (TDD). By contrast, nodes in the TDD representation correspond to indices in the circuit (regarded as a tensor network) and each node has two child nodes according to their edges which are combined multiplicatively.

**A. Basic Definition**

To begin with, we observe that any tensor $\phi$ can be expanded with respect to a given index in the style of Boole-Shannon expansion for classical Boolean circuits.

**Lemma 1.** Let $\phi$ be a tensor with indices in $I$. For each $x \in I$, 
\[
\phi = \mathbb{I} \cdot \phi|_{x=0} + x \cdot \phi|_{x=1},
\] (5)

**Definition 1** (Tensor Decision Diagram). A Tensor Decision Diagram (TDD) $\mathcal{F}$ over a set of indices $I$ is a rooted, weighted, and directed acyclic graph $\mathcal{F} = (V, E, index, value, low, high, w)$ defined as follows:

- $V$ is a finite set of nodes which is partitioned into non-terminal nodes $V_N$ and terminal ones $V_T$. Denote by $r_\mathcal{F}$ the unique root node of $\mathcal{F}$;
- $index : V_N \rightarrow I$ assigns each non-terminal node an index in $I$;
- $value : V_T \rightarrow \mathbb{C}$ assigns each terminal node a complex value;
- both low and high are mappings in $V_N \rightarrow V$ which assign each non-terminal node with its 0- and 1-successors, respectively;
- $E = \{(v, low(v)), (v, high(v)) : v \in V_N\}$ is the set of edges, where $(v, low(v))$ and $(v, high(v))$ are called the low- and high-edges of $v$, respectively. For simplicity, we also assume the root node $r_\mathcal{F}$ has a unique incoming edge, denoted $e_r$, which has no source node;
- $w : E \rightarrow \mathbb{C}$ assigns each edge a complex weight. In particular, $w(e_r)$ is called the weight of $\mathcal{F}$, and denoted $w_\mathcal{F}$.

A TDD is called trivial if its root node is also a terminal node.

For convenience, we often call a terminal node with value $c$ a terminal $c$ node or simply terminal $c$ if it is unique.

The following example shows how a tensor can be transformed to a TDD using the Boole-Shannon expansion.

**Example 4.** Fig. 6 gives the TDD obtained by directly applying the Boole-Shannon expansion to the tensor $\phi_{x_0,x_0,x0,x0}^4$ in Eq. 4 where and in all illustrations in this paper we omit the weight of an edge if it is $1$. Each terminal node bears a value which, when multiplying with weights along the path to the root node (which happen to be all 1 in this example), corresponds to the value of $\phi$ under the evaluation specified by the path. For example, the terminal node with value $1$ corresponds to the value of $\phi$ under the evaluation $\{x_0 \mapsto 1, y_0 \mapsto 0, y_3 \mapsto 0\}$. Each non-terminal node $v$ acts as a decision node and represents...
an index \( x \), while its low- and high-edges denote evaluations which evaluate \( x \) to 0 and, respectively, 1.

Conversely, let us see how each node \( v \) of a TDD \( F \) naturally corresponds to a tensor \( \Phi(v) \). If \( v \) is a terminal node, then \( \Phi(v) := \text{value}(v) \) is a rank 0 tensor, i.e., a constant; if \( v \) is a non-terminal node, then

\[
\Phi(v) := w_0 \cdot \pi_v \cdot \Phi(\text{low}(v)) + w_1 \cdot x_v \cdot \Phi(\text{high}(v)),
\]

where \( x_v = \text{index}(v) \), and \( w_0 \) and \( w_1 \) are the weights on the low- and high-edges of \( v \), respectively. Comparing Eq. 6 with the Boole-Shannon expansion in Lemma 1, we immediately have

\[
\Phi(v)|_{x_v=c} = w_c \cdot \Phi(v_c),
\]

where \( c \in \{0,1\} \), \( v_0 = \text{low}(v) \), and \( v_1 = \text{high}(v) \).

Finally, the tensor represented by \( F \) itself is defined to be

\[
\Phi(F) := \Phi(r_F).
\]

Recall here that \( r_F \) and \( w_F \) are the root node and the weight of \( F \), respectively.

An efficient manipulation of general TDDs seems impossible. Following [7], we restrict our discussion to ordered TDDs.

**Definition 2.** A TDD \( F \) is called ordered if there is a linear order \( \prec \) on \( I \) such that \( \text{index}(v) \prec \text{index}(\text{low}(v)) \) and \( \text{index}(v) \prec \text{index}(\text{high}(v)) \) for every non-terminal node \( v \), provided that both \( \text{low}(v) \) and \( \text{high}(v) \) are non-terminal as well. If this is the case, we say \( F \) is a \( \prec \)-ordered TDD.

For simplicity, we abuse the notation slightly by assuming \( x \prec \text{index}(v) \) for all \( x \in I \) and all terminal nodes \( v \in V_T \).

The size of a TDD \( F \), written \( \text{size}(F) \), is the number of non-terminal nodes of \( F \). As each non-terminal node has two outgoing edges, there are altogether \( 1 + 2 \times \text{size}(F) \) edges, including \( e_r \), in \( F \). Like ROBDDs, the size of the TDD representation strongly relies on the selected variable order. For example, the tensor \( \phi = \langle x_1 \cdot x_2 \rangle + \langle x_3 \cdot x_4 \rangle + \langle x_5 \cdot x_6 \rangle \) can be represented as a TDD with 6 non-terminal nodes under the order \( \prec_1 := \langle x_1, x_2, x_3, x_4, x_5, x_6 \rangle \), but its TDD representation under \( \prec_2 := \langle x_1, x_3, x_5, x_2, x_4, x_6 \rangle \) requires at least \( 2 \times (1 + 2^1 + 2^2) = 14 \) internal nodes (cf. [20] Ch.3).

While finding an optimal order is NP-hard, there are efficient heuristic methods that have been devised for ROBDDs, which may also be extended to TDDs.

**B. Normalisation**

A tensor may have many different TDD representations. For example, let \( F \) be a TDD with root node \( r_F \) and weight \( w_F \neq 0 \). A different TDD representing the same tensor can be constructed by, for example, multiplying \( w_F \) by 2 and dividing the weights of the low- and high-edges of \( r_F \) by 2. In order to provide a canonical representation, we first introduce the notion of normal tensors.

**Definition 3 (normal tensor).** Let \( \phi \) be a tensor with index set \( I = \{ x_1, \ldots, x_n \} \) and \( \prec \) a linear order on \( I \). We write

\[
\| \phi \| := \max_{\vec{a} \in \{0,1\}^I} |\phi(\vec{a})|
\]

for the maximum norm of \( \phi \). Let \( \vec{a}^* \) be the first element in \( \{0,1\}^I \) (under the lexicographical order induced by \( \prec \)) which has the maximal magnitude under \( \phi \), i.e.,

\[
\vec{a}^* = \min\{ \vec{a} \in \{0,1\}^I : |\phi(\vec{a})| = \| \phi \| \}.
\]

We call \( \vec{a}^* \) the pivot of \( \phi \). A tensor \( \phi \) is called normal if either \( \phi = 0 \) or \( \phi(\vec{a}^*) = 1 \).

It is easy to see that there are tensors \( \phi \) with \( \| \phi \| = 1 \) but \( \phi \) is not normal. The following lemma shows that any tensor can be normalised in a unique way.

**Lemma 2.** For any tensor \( \phi \) which is not normal, there exists a unique normal tensor \( \phi^* \) such that \( \phi = p \cdot \phi^* \), where \( p \) is a nonzero complex number.

The uniqueness of the normal tensor in the above lemma suggests the following definition.

**Definition 4.** A TDD \( F \) is called normal if \( \Phi(v) \) is a normal tensor for every node \( v \) in \( F \).

It is worth noting that as normal TDDs may still have arbitrary weights, tensors represented by normal TDDs do not have to be normal. Normal TDDs enjoy some nice properties collected in the following two lemmas.

**Lemma 3.** Suppose \( F \) and \( G \) are two normal TDDs such that \( \Phi(F) = \Phi(G) \). Then we have \( w_F = w_G \) and \( \Phi(r_F) = \Phi(r_G) \).

**Proof.** By Eq. 8 we have \( \Phi(F) = w_F \cdot \Phi(r_F) \) and \( \Phi(G) = w_G \cdot \Phi(r_G) \). Because \( \Phi(r_F) \) and \( \Phi(r_G) \) are normal tensors and \( \Phi(F) = \Phi(G) \), by Lemma 2 we know \( w_F = w_G \) and \( \Phi(r_F) = \Phi(r_G) \).

For any non-normal TDD \( F \), we can transform it into a normal one by applying the following two rules.

**Normalisation Rules.**

**NR1:** If \( v \) is a terminal node with a nonzero value \( \text{value}(v) \neq 0 \), then set its value to 1, and change the weight \( w \) of each incoming edge of \( v \) to \( \text{value}(v) \cdot w \).

**NR2:** Suppose \( v \) is a non-terminal node such that \( \Phi(v) \neq 0 \) is not normal but both \( \Phi(\text{low}(v)) \) and \( \Phi(\text{high}(v)) \) are normal. Let \( w_0 \) and \( w_1 \) be the weights on the low- and high-edges of \( v \) respectively. If \( \Phi(\text{low}(v)) \neq 0 \) and either \( \Phi(\text{high}(v)) = 0 \) or \( |w_0| \geq |w_1| \), we set \( w \) to be \( w_0 \). Otherwise, set it to be \( w_1 \). Divide \( w_0 \) and \( w_1 \) by \( w \) and multiply the weight of each incoming edge of \( v \) by \( w \).

Let \( F \) be a non-normal TDD. We first apply \( \text{NR1} \) to every terminal node of \( F \) to make it normal. Furthermore, if a non-terminal node \( v \) of \( F \) represents a non-normal tensor but both its successors represent normal tensors, then, it is easy to see that after applying \( \text{NR2} \) to \( v \), this node itself represents a normal tensor. This gives a procedure to transform \( F \) into a normal TDD in a bottom-up manner. Furthermore, the transformation can be done within time linear in the size of \( F \).

**Theorem 1.** Applying a normalisation rule to a TDD does not change the tensor it represents. Moreover, a TDD is normal if and only if no normalisation rule is applicable.
Fig. 7 (c), is obtained by applying NR2 to the right

Theorem 2. Let $≺$ exists a

C. Reduction

As can be seen from Fig. 7, normal TDDs may still have redundant nodes. For example, the first and the third $y_3$ nodes of the normal TDD in Fig. 7(c) have the same low- and high-edges and thus represent the same tensor. This fact motivates us to further introduce:

Definition 5. A TDD $\mathcal{F}$ is called reduced if it is normal and

1) no node represents the 0 tensor, i.e., $\Phi(v) \neq 0$ for every node $v$ in $\mathcal{F}$;
2) all edges weighted 0 point to the (unique) terminal 1; and
3) no two different nodes represent the same tensor, i.e., $\Phi(u) \neq \Phi(v)$ for any two nodes $u \neq v$ in $\mathcal{F}$.

The following lemma shows that every non-terminal node of a reduced TDD $\mathcal{F}$ is labelled with an essential variable of the tensor represented by $\mathcal{F}$.

Lemma 4. Suppose $\mathcal{F}$ is a reduced TDD of a non-constant tensor $\phi$ over index set $I$. Then every non-terminal node of $\mathcal{F}$ is labelled with an index that is essential to $\phi$.

The following definition of sub-TDDs is useful in our later discussion. Recall that we assume $x \prec \text{index}(v)$ for all $x \in I$ and all terminal nodes $v$.

Definition 6. Let $\mathcal{F}$ be a reduced TDD over a $\prec$-linearly ordered index set $I$. Let $x \in I$, and $x \preceq \text{index}(r_{\mathcal{F}})$. We define sub-TDDs $\mathcal{F}_{x=0}$ and $\mathcal{F}_{x=1}$ of $\mathcal{F}$ as follows.

1) If $x \prec \text{index}(r_{\mathcal{F}})$, then $\mathcal{F}_{x=0} = \mathcal{F}_{x=1} = \mathcal{F}$;
2) If $x = \text{index}(r_{\mathcal{F}})$, $\mathcal{F}_{x=0}$ is defined as the TDD rooted at low($r_{\mathcal{F}}$) with weight $w(r_{\mathcal{F}}) \cdot w(r_{\mathcal{F}}, \text{low}(r_{\mathcal{F}}))$, i.e., the weight of the low-edge of $r_{\mathcal{F}}$ multiplied by the weight of $\mathcal{F}$. Analogously, we have $\mathcal{F}_{x=1}$.

Corresponding to the Boolean-Shannon expansion for tensors (cf. Eq. 5), we have

Lemma 5. Suppose $\mathcal{F}$ is a reduced TDD on $I$, $x \in I$ and $x \preceq \text{index}(r_{\mathcal{F}})$. Then we have

$$\Phi(\mathcal{F}) = x \cdot \Phi(\mathcal{F}_{x=0}) + \Phi(\mathcal{F}_{x=1}).$$

Now we are ready to prove the canonicity of reduced TDDs. Two TDDs $\mathcal{F}$ and $\mathcal{G}$ are said to be isomorphic, denoted $\mathcal{F} \equiv \mathcal{G}$, if they are equal up to renaming of the nodes; that is, there exists a graph isomorphism between $\mathcal{F}$ and $\mathcal{G}$ which preserves node indices, edge weights, and values on terminal nodes. Furthermore, it maps low-edges to low-edges and high-edges to high-edges.

Theorem 3 (canonicity). Let $I$ be an index set and $\prec$ a linear order on $I$. Suppose $\mathcal{F}$ and $\mathcal{G}$ are two $\prec$-ordered, reduced TDDs over $I$ with $\Phi(\mathcal{F}) = \Phi(\mathcal{G})$. Then $\mathcal{F} \equiv \mathcal{G}$.

A reduced TDD can be obtained by applying the following reduction rules on any normal TDD in a bottom-up manner.

Reduction rules.

RR1: Merge all terminal 1 nodes. Delete all terminal 0 ones, if exist, and redirect their incoming edges to the (unique) terminal and reset their weights to 0.
Applying RR1 to merge all terminal 1 nodes and delete all nodes (as well as all edges involving them) which are not reachable from the root node.

RR3: Delete a node $v$ if its 0- and 1-successors are identical and its low- and high-edges have the same weight $w$ (either 0 or 1). Meanwhile, redirect its incoming edges to terminal 1 if $w = 0$ and, if otherwise, to its successor.

RR4: Merge two nodes if they have the same index, the same 0- and 1-successors, and the same weights on the corresponding edges.

**Theorem 4.** A normal TDD is reduced if and only if no reduction rule is applicable.

Theorem 5. Let $F$ be a normal TDD representing tensor $\phi$. Applying a reduction rule to $F$ does not change the tensor it represents. Moreover, the reduced TDD of $\phi$ can be obtained from $F$ by applying the reduction rules till no one is applicable.

**Proof.** It is routine to show that applying any reduction rule to a normal TDD does not change the tensor it represents. Suppose $F$ is a normal TDD that is not reduced. Applying the reduction rules in a bottom-up manner until no rule is applicable, by Theorem 4 we obtain a reduced TDD that also represents $\phi = \Phi(F)$. As reduced TDDs are unique (see Theorem 3), this gives the reduced TDD of $\phi$. □

As each application of a reduction rule removes some nodes, the reduced TDD has the minimal number of nodes.

**Corollary 1.** Let $F$ be a normal TDD of a tensor $\phi$. Then $F$ is reduced if and only if $\text{size}(F) \leq \text{size}(G)$ for all normal TDDs of $\phi$.

**Example 6.** Consider the normalised TDD shown in Fig. 7(c). Applying RR1 to merge all terminal 1 nodes and delete all terminal 0 nodes gives the TDD shown in Fig. 8(a). Then, further applying RR4 to merge the first and the third as well as the second and the fourth $y_3$ nodes, we have the reduced TDD as shown in Fig. 8(b), which provides a compact representation for the circuit in Fig. 4.

**Remark 1.** As Boolean functions are special tensors, each Boolean function also has a unique reduced TDD representation, which can be obtained by performing the reduction rule RR1 on its ROBDD representation if we assign weight 1 to each ROBDD edge.

V. ALGORITHMS

This section is devoted to algorithms for constructing the corresponding reduced TDD from a given tensor and key operations such as addition and contraction of TDDs. All of these algorithms are implemented in a recursive manner. Every time a new node is generated, we apply normalisation and reduction rules locally to this node, implemented by calling the reduce procedure. In this way, it can be guaranteed that the TDDs obtained are all reduced. It is also worth noting that motivated by [6], to avoid redundancy, in our real implementation (not shown in the algorithms) all the nodes are stored in a hash table. Whenever a new node is about to be generated, we first check if such a node (with the same index, successors and weights on the corresponding edges) already exists in the table. If yes, the node is returned directly; otherwise, a new one is created and added into the hash table.

**A. Generation**

Algorithm 1 shows the process of generating the reduced TDD of a tensor. The time complexity of the construction is linear in $|V|$, the number of nodes in the constructed TDD.

**Algorithm 1 TDD_generate($\phi$)**

**Input:** A tensor $\phi$ over a linearly ordered index set $I$.

**Output:** The reduced TDD of $\phi$.

1. if $\phi \equiv c$ is a constant then
2. return the trivial TDD with weight $c$
3. end if
4. $x \leftarrow$ the smallest index of $\phi$
5. $tdd \leftarrow$ an empty TDD
6. $tdd.root \leftarrow$ a new node $v$ with index $x$
7. $v.low \leftarrow$ TDD_generate($\phi|_{x=0}$)
8. $v.high \leftarrow$ TDD_generate($\phi|_{x=1}$)
9. $tdd.weight \leftarrow 1$
10. return reduce(tdd)

We emphasise that, if an index is repeated in the tensor, for example $\phi_{xxy}$, then the two successors of the node representing $\phi_{xxy}$ will be $\phi_{00y}$ and $\phi_{11y}$. In other words, we construct the TDD as if it is the tensor $\phi_{xxy}$. When tensor operations are concerned, however, both $x$ indices will be involved.

**Example 7.** Consider the $C_{X}$ gate shown in Fig. 5 which is represented by a tensor $\phi_{x0,x0,y1,y2}$. The reduced TDD of $\phi_{x0,x0,y1,y2}$ is shown in Fig. 7(b), where the index $x_0$ only appears once with the two successors representing the tensor $\phi_{00,y1,y2}$ and $\phi_{11y1,y2}$.

**B. Addition**

Let $F$ and $G$ be two reduced TDDs over index set $I$. The summation of $F$ and $G$, denoted $F + G$, is a reduced TDD...
with the corresponding tensor $\Phi(F) + \Phi(G)$. For any $x \in I$ with $x \preceq \text{index}(r_F)$ and $x \preceq \text{index}(r_G)$, by the TDD version of the Boole-Shannon expansion (cf. Eq. \ref{eq:ts}), we have
\[
\Phi(F) + \Phi(G) = \pi \cdot (\Phi(F_{x=0}) + \Phi(G_{x=0})) + x \cdot (\Phi(F_{x=1}) + \Phi(G_{x=1})).
\]

Recall here $F_{x=c}$ (resp. $G_{x=c}$) is the sub-TDD as defined in Definition \ref{def:TDD} for $c \in \{0, 1\}$.

Motivated by this observation, Algorithm \ref{alg:Add} implements the Add operation for TDDs, in a node-wise manner. The time complexity is $O(|F| \cdot |G|)$, where $|F|$ and $|G|$ denote the numbers of nodes in the two TDDs respectively.

\begin{algorithm}
\caption{Add($F$, $G$)}
\begin{algorithmic}[1]
\Input{Two reduced TDDs $F$ and $G$.}
\Output{The reduced TDD of $\Phi(F) + \Phi(G)$.}
\Function{Add}{F, G}
\If{$r_F = r_G$}
\State $tdd \leftarrow F$
\State $tdd.weight \leftarrow w_F + w_G$
\State \textbf{return} tdd
\EndIf
\State $x \leftarrow$ the smaller index of $r_F$ and $r_G$
\State $tdd \leftarrow$ an empty TDD
\State $tdd.root \leftarrow$ a new node $v$ with index $x$
\State $v.low \leftarrow \text{Add}(F_{x=0}, G_{x=0})$
\State $v.high \leftarrow \text{Add}(F_{x=1}, G_{x=1})$
\State $tdd.weight \leftarrow 1$
\State \textbf{return} reduce(tdd)
\EndFunction
\end{algorithmic}
\end{algorithm}

\section{Contraction}

Contraction is the most fundamental operation in a tensor network. Many design automation tasks of quantum circuits are based on contraction. In this subsection, we consider how to efficiently implement the contraction operation via TDD.

Let $F$ and $G$ be two reduced TDDs over $I$, and $var$ a subset of $I$ denoting the variables to be contracted. Write $\text{cont}$ for both tensor and TDD contractions. For any $x \in I$ with $x \preceq \text{index}(r_F)$ and $x \preceq \text{index}(r_G)$, we have by definition Eq. \ref{eq:ts} that if $x \in var$, then $\text{cont}(\Phi(F), \Phi(G), var)$ equals
\[
\sum_{c=0}^{1} \text{cont}(\Phi(F_{x=c}), \Phi(G_{x=c}), var \setminus \{x\});
\]
otherwise, it equals
\[
\pi \cdot \text{cont}(\Phi(F_{x=0}), \Phi(G_{x=0}), var) + x \cdot \text{cont}(\Phi(F_{x=1}), \Phi(G_{x=1}), var).
\]

Algorithm \ref{alg:contr} gives the detailed procedure for TDD contraction. The time complexity is $O(|F|^2 \cdot |G|^2)$, while $|F|$ and $|G|$ denote the numbers of nodes in $F$ and $G$, respectively.

To conclude this section, we would like to point out that the tensor product of two TDDs $F$ and $G$ with disjoint essential indices can be regarded as a special case of contraction. In particular, we have
\[
\Phi(F \otimes G) = \text{cont}(\Phi(F), \Phi(G), \emptyset),
\]
and the time complexity of using Algorithm \ref{alg:contr} to compute $F \otimes G$ becomes $|F| \cdot |G|$.

A special case which arises often in applications is when, say, every index in $F$ precedes any index in $G$ under the order $\prec$. For this case, to compute the tensor product of $F$ and $G$, all we need to do is to replace the terminal node of $F$ with the root node of $G$, multiply the weight of the resulting TDD with the weight of $G$, and perform normalisation and reduction if necessary. Since we do not need to touch $G$, the time complexity is simply $O(|F|)$.

\begin{algorithm}
\caption{cont($F$, $G$, var)}
\begin{algorithmic}[1]
\Input{Two reduced TDDs $F$ and $G$, and the set var of variables to be contracted.}
\Output{The reduced TDD obtained by contracting $F$ and $G$ over var.}
\Function{cont}{$F$, $G$, var}
\If{both $F$ and $G$ are trivial}
\State $tdd \leftarrow F$
\State $tdd.weight \leftarrow w_F \cdot w_G \cdot 2^{|\text{len}(var)|}$
\State \textbf{return} tdd
\EndIf
\State $x \leftarrow$ the smaller index of $r_F$ and $r_G$
\State $tdd \leftarrow$ an empty TDD
\State $tdd.root \leftarrow$ a new node $v$ with index $x$
\State $v.low \leftarrow \text{Add}(F_{x=0}, G_{x=0})$
\State $v.high \leftarrow \text{Add}(F_{x=1}, G_{x=1})$
\State $tdd.weight \leftarrow 1$
\State \textbf{return} reduce(tdd)
\EndFunction
\end{algorithmic}
\end{algorithm}

\section{Two partition schemes}

The TDD representation of a quantum circuit can be calculated flexibly. In particular, there is no need to expand a quantum gate to an $n$-qubit form (by tensoring an identity matrix). In general, the TDD representation of a quantum circuit can be obtained by contracting the TDDs of individual gates in the circuit in any order.
In this paper, we assume that all gates in our circuits are either single-qubit gates or CX gates. For simplicity of presentation, we use the original qubit order (or its inverse). Following this order, we scan the circuit qubit by qubit, and then rank the indices following the qubit order. That is, given two indices \( x \) and \( x' \) appearing in the circuit, suppose \( q_i \) and \( q_j \) are the qubits that \( x \) and \( x' \) are on. Then we set \( x < x' \) if either \( i < j \), or \( i = j \) and \( x \) is to the left of \( x' \) on the qubit wire \( q_i \). For example, the selected order for the circuit shown in Fig. [10] is

\[
(x_0, x_{0,1}, x_{0,2}, y_0, x_1, x_{1,1}, x_{1,2}, y_1, x_2, \ldots, y_2, x_3, \ldots, y_3).
\]

Our approach of computing the TDD of a quantum circuit includes two steps. First, we partition the circuit into several parts; second, we calculate the TDD of each part separately and then combine them together through contraction.

While finding the optimal partition scheme is attractive, it is also a very challenging task. We observe that some simple strategies are already able to reduce the resource consumption significantly during the computation process. In the following, we introduce two straightforward partition schemes.

The first partition scheme divides the circuit horizontally into two parts from the middle (so that the upper and lower parts have roughly the same number of qubits) and then cuts it vertically such that in each part no more than a \( k \) (a predefined parameter) CX gates are separated by the horizontal cut, where \( k \) is chosen to ensure that the rank of each block of the final circuit is smaller than \( 2n \), the rank of the tensor of the original circuit. In our experiments, we set \( k = \lfloor n/2 \rfloor \).

**Example 8.** Consider the circuit given in Fig. [10] again. Suppose we allow one CX cut every time, and limit the number of qubits in part \( C \) to two. Then the circuit can be partitioned into five parts as illustrated in Fig. [11]. We then compute and contract the TDDs in the order of \( \mathcal{A}, \mathcal{B}, \mathcal{C} \) for every block split by the vertical lines. The TDD of the whole circuit is then obtained by contracting the TDDs of these blocks in sequence.

Now we make a simple comparison of the above contraction methods. Suppose we compute the TDD (or QMDD) representation of the circuit in Fig. [10] in the original circuit order. We need in essence to calculate eight (8, 2, 1)-contractions, five (8, 4, 2)-contractions, one (6, 2, 0)-contraction, and two contractions between tensors with rank \( \leq 4 \), where an \((m, n, r)\)-contraction is a contraction between a rank \( m \) tensor and a rank \( n \) tensor over \( r \) common indices. In comparison, Partition Scheme I requires one (8, 8, 4)-contraction, two (5, 5, 1)-contractions, five (5, 2, 1)-contractions, and nine contractions between tensors with rank \( \leq 4 \); while Partition Scheme II requires one (8, 8, 4)-contraction, one (8, 4, 2)-contraction, one (5, 5, 1)-contraction, and 14 contractions between tensors with rank \( \leq 4 \). As the time and space consumption both grow exponentially with the ranks of the tensors [13], this illustrates the efficiency of the two partition schemes.

VII. IMPLEMENTATION AND EVALUATION

To demonstrate the effectiveness of TDD as an efficient data structure for the representation and manipulation of quantum functionalities, we developed the TDD package in Python3, im-

---

*Fig. 10. Partition Scheme I, where only one CX cut is allowed each time.*

*Fig. 11. Partition Scheme II, where only one CX cut is allowed each time and part \( C \) involves up to 2 qubits.*
implemented the two partition schemes, and empirically compared them with three state-of-the-art approaches in the literature.

A. Benchmarks

Most benchmarks we used were published by IBM as part of the 2018 QISKit Developer Challenge[1], which have been wildly used in evaluating qubit mapping algorithms (see, e.g., [28]). To compare the scalability of different methods, we also tested three commonly used quantum algorithms, including Bernstein-Vazirani (bv) [3]. Quantum Fourier Transform (qft) [22], as well as Quantum Volume (qv) [21]. The numbers of qubits and gates in these benchmarks range from 2 to 100 and 5 to 10,223, respectively.

B. TDD Implementation

We implemented TDD using Python3. In our calculation process, calculated results are stored in a computed table as in the implementation of ROBDD [6]. In order to improve the reusability of the calculated results in the computed table, we map all indices of a TDD to \{0,1,\ldots,m−1\}, where \(m\) is the number of different indices of the tensor associated to the TDD, such that TDDs differ only by a renaming of indices will be treated as the same. Our source code is available at Github[2].

In our experiments, for the first partition scheme, we set the parameter \(k\) as \(\lceil n/2 \rceil\), where \(n\) is the number of qubits in the input circuit. Similarly, for the second partition scheme, we set the two parameters \(k_1\) and \(k_2\) as \(\lceil n/2 \rceil\) and \(\lceil n/2 \rceil + 1\), respectively. All experiments were executed on a laptop with Intel i7-1065G7 CPU and 8 GB RAM.

C. Empirical results

We compared our results with three state-of-the-art approaches for computing quantum functionalities — Qiskit (https://qiskit.org), the Google TensorNetwork package [25], and QMDD [23]. The first two approaches are matrix-based, while QMDD is decision diagram-based. For Qiskit, we call the unitary_simulator for calculating the unitary matrix of every circuit, and for TensorNetwork, a tensor network is constructed for every circuit and the auto contractor will be used for completing the task. For QMDD, we compute the functionality of an input circuit in a way similar to TDD with no partition, i.e., we construct the QMDD of each quantum gate and then multiply them in the circuit order. The QMDD package we used is the version obtained from http://www.informatik.uni-bremen.de/agra/eng/qmdd.php.

We summarise our experimental results on the benchmark circuits from [28] in Table I. More and detailed results can be found in Table II of the Appendix. The performance of Qiskit is very similar but inferior to TensorNetwork. Except the 15 qubit circuit ‘rd84_142’, Qiskit can finish in 116.7s all circuits which TensorNetwork does not run out of memory.

For ‘rd84_142’, Qiskit runs out of memory but TensorNetwork finishes in 49.3s. In the following, we omit the results of Qiskit from the table.

1) Compare with matrix-based methods: As mentioned before, matrix-based methods, like Qiskit and TensorNetwork, represent an \(n\)-qubit circuit by a \(2^n \times 2^n\) matrix. Assume that all data in such a matrix is represented in data type complex128. Then 64GB of memory must be allocated for the matrix of a 16-qubit circuit. This implies in particular that in our laptop (with 8GB RAM) these methods can process quantum circuits with at most 15 qubits. This observation is confirmed by our experiments. In comparison, the DD-based methods are often much more compact. Indeed, for the qft circuits and on our laptop, both QMDD and TDD can easily process quantum circuits with up to 21 qubits. For example, both DDs can generate the functionality of the circuit ‘qft_21’ by using less than 128 MB memory. For the bv circuits, this characteristic of DDs is even striking, as both DDs can easily process bv circuits with as many as 100 qubits in a few seconds!

On the other hand, when the number of qubits is small \((\leq 10)\), TensorNetworking usually works faster than DD-based methods. This is perhaps due to that the time-consumption for transforming matrices to decision diagrams is not negligible. Surprisingly, while it takes only 0.08s for TensorNetwork to compute the tensor of the circuit ‘qv_n9_d5’ (with 9 qubits), both QMDD and TDD time out. It turns out that the TDD representation has 262,144 nodes, while in comparison the TDD of ‘qv_n9_d5’ has 262,143 nodes. This suggests that some quantum circuits can be better processed by TensorNetwork than DD-based methods and there are quantum circuits which may have no compact DD representations.

2) Compare with QMDD: Recall that every non-terminal node in a TDD has two successors while any non-terminal node in a QMDD has four. In principle, the TDD representation of a quantum circuit has about twice the number of nodes as the circuit’s QMDD, provided that the same order is used. If this is the case, the memory usage of the TDD representation is roughly the same as that of the QMDD representation. This is because they have the same number of weighted edges and store the same number of weights (complex numbers). This observation is consistent with our experimental results. As a consequence, the TDD representation is as compact as QMDD.

As far as runtime efficiency is concerned, on the benchmarks from [23], the runtime of the three TDD schemes are, respectively, 2.9, 3.8, 8.4 times of that of QMDD; but, if including all the bv, qv, and qft circuits we have tested, the runtime of

---

1https://www.ibm.com/blogs/research/2018/08/winners-qiskit-developer-challenge/
2Available from http://iic.jku.at/eda/research/ibm_qx_mapping/
3https://github.com/VeriQC/TDD

---

TABLE I

|   | QMDD | TDD | TN |
|---|------|-----|-----|
| Time (−MO) | 30.66 | 259.38 | 117.51 |
| node num. (final*) | 6413 | 36194 | 17769 |
| node num. (max*) | 15758 | 13888 | 13888 |
| ratio (max/final*) | 2.46 | 2.61 | 1.28 |

*node num. (final*) denotes the total number of nodes in the computed table, node num. (max*) denotes the maximum number of nodes in the computed table, ratio (max/final*) denotes the ratio of the maximum number of nodes to the final number of nodes. Time (−MO) denotes the total time of all circuits on which TN (the Google TensorNetwork) is not memory out.
QMDD is about 1.3, 3.1, 3.5 times of that of the three TDD schemes. Considering that the TDD package is implemented in Python and QMDD is implemented in C++, this suggests that TDD is at least comparable with QMDD. Moreover, there are quantum circuits in which the TDD package outperforms QMDD. As can be observed from Fig. 12 for qft circuits with 17 or more qubits, the no-partition scheme of TDD is already faster than QMDD. Even better, TDD with either partition scheme can process the qft-22 circuit with around 1500s while QMDD times out.

3) Compare among TDD schemes: In general, the two partition schemes can significantly decrease the time-consumption for constructing the functionality of a quantum circuit. From Table I we can see that both partition schemes can decrease the time-consumption by at least 50% when compared with the no-partition scheme. This judgement is also confirmed by experiments on bv, qft, and qv circuits (cf. Table II of the appendix).

Table I also suggests that the TDD construction with either partition scheme often has smaller intermediate diagrams than QMDD and the no-partition TDD. Let
\[ \alpha = \frac{\text{maximum size of all DDs during the computing process}}{\text{the size of the final DD}}. \]

Table I shows that the \( \alpha \) values of QMDD and the three TDD schemes are, respectively, 2.46, 2.61, 1.28 and 1.25. That is, the ratio could be halved if either partition scheme is adopted.

D. Summary and Discussion

From the above empirical results we can see that
- TDD is compact and memory-saving and can be used for calculating the functionality of large circuits.
- TDD is flexible and can easily be combined with tensor network techniques (e.g., partition) to further improve its performance.

Besides representing the functionality of quantum circuits, TDD can also be used in the classical simulation of quantum circuits. Experimental results show that the performance of TDD is similar to that of QMDD reported in [29]. For example, we can obtain all amplitudes of ‘qft\(_k\)’ circuits within 3 seconds for \( k \leq 64 \). We also conducted experiments on the simulation of random quantum circuits. The performance is also similar to that of QMDD.

In addition, TDD can also be used to calculate the trace of a quantum circuit, which plays a central role in calculating fidelity and hence checking if two quantum circuits are approximately equivalent [15]. As trace calculation is a more tensor network fit task, TDD could be more convenient for such a task.

As a direct extension of BDD from Boolean functions to tensors, TDDs can also represent classical gates. More important, we can also represent the measurements and classically controlled gates as TDDs, which makes it suitable for coping with tasks such as equivalence checking of dynamic quantum circuits [14].

VIII. Conclusion

We proposed a decision diagram style data structure — TDD — for more principled applications of tensor networks. Based on a Boole-Shannon style expansion for tensors, it is rigorously proved that TDD provides a universal and canonical representation for quantum functionalities. As a decision diagram, TDD is also compact as redundant or isomorphic nodes have been completely removed or merged. Experiments on a variety of benchmark circuits include qft confirm that TDD is compact, demonstrate its efficiency, and show that it often outperforms the Google TensorNetwork package for circuits with 15 or more qubits. Moreover, thanks to its origin from tensor networks, many techniques developed or to be developed for tensor networks can be directly imported into TDD. As an example, we have shown that the TDD of a quantum circuit can be computed more efficiently by exploiting circuit partitions that were previously introduced for the classical simulation of quantum circuits.

It is expected that TDD can be used, possibly together with the Google TensorNetwork, in many design automation tasks, e.g., simulation and equivalence checking, for quantum circuits. In particular, we plan to combine TDD with TensorNetwork in our partition-based schemes. When the rank of the tensor is small, TensorNetwork runs faster than both decision diagrams; however, its performance decrease sharply when the number of qubits increases. Thus, we can use TensorNetwork to compute the (local) functionalities of each part, transform them into TDDs, and then contract these local TDDs to obtain the TDD representation of the quantum circuit.

The current TDD package is far from being optimal. Future work will implement TDD in C++ and try to exploit more optimisation techniques developed in tensor networks, e.g., tree decomposition [18].

In this paper, we assume that all indices of a tensor can only take values from \( \{0, 1\} \). This restriction can be removed by allowing a node in a TDD to have any number of successors. Moreover, different nodes can even have different numbers of successors. What we should ensure is that all nodes corresponding to the same index have the same number of successors, and the contraction should be conducted on all its successors when this index is contracted. In our follow-up work, we plan to construct such a generalised package and make it suitable for more tensor network tasks.
REFERENCES

[1] ABUTTE, F., ARYA, K., BABBUSSH, R., BACON, D., BARDIN, J. C., BARENS, R., BISWAS, R., BOIXO, S., BRANDAO, F. G., BUELL, D. A., ET AL. Quantum supremacy using a programmable superconducting processor. Nature 574, 7779 (2019), 505–510.

[2] BAHRAH, R. I., FORQUÉ, E. A., GAONA, C. M., HACHTHEL, G. D., MACII, E., PARDO, A., AND SOMENZI, F. Algorithmic decision diagrams and their applications. Formal methods in system design 10, 2-3 (1997), 171–206.

[3] BERNESTEIN, E., AND VAZIRANI, U. Quantum complexity theory. SIAM Journal on Computing 26, 5 (1997), 1411–1473.

[4] BIAMONTE, J. Lectures on quantum tensor networks. arXiv preprint arXiv:1912.10049 (2019).

[5] BOIXO, S., ISAKOV, S. V., SMELYANSKIY, V. N., AND NEVEN, H. Simulation of low-depth quantum circuits as complex undirected graphical models. arXiv preprint arXiv:1712.05384 (2017).

[6] BRACE, K. S., RUDELL, R. L., AND BRYANT, R. E. Efficient implementation of a bdd package. In 27th ACM/IEEE design automation conference (1990), IEEE, pp. 40–45.

[7] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on 100, 8 (1986), 677–691.

[8] BRYANT, Y.-A. C. R. E Verification of arithmetic circuits with binary moment diagrams. In 32nd Design Automation Conference (1995), IEEE, pp. 535–541.

[9] BURGHOHLER, L., AND WILLI, R. Advanced equivalence checking for quantum circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[10] BURGHOHLER, L., AND WILLI, R. Improved dd-based equivalence checking of quantum circuits. In 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC) (2020), IEEE, pp. 127–132.

[11] CHEN, J., ZHANG, F., HUANG, C., NEWMAN, M., AND SHI, Y. Classical simulation of intermediate-size quantum circuits. arXiv preprint arXiv:1805.01450 (2018).

[12] CHEN, Z.-Y., ZHOU, Q., XUE, C., YANG, X., GUO, G.-C., AND GUO, G.-P. 64-qubit quantum circuit simulation. Science Bulletin 63, 15 (2018), 964–971.

[13] GRAY, J., AND KOURTIS, S. Hyper-optimized tensor network contraction. arXiv preprint arXiv:2002.01956 (2020).

[14] HONG, X., FENG, Y., LI, S., AND YING, M. Equivalence checking of dynamic quantum circuits. arXiv preprint arXiv:2106.01658 (2021).

[15] HONG, X., YING, M., FENG, Y., ZHOU, X., AND LI, S. Approximate equivalence checking of noisy quantum circuits. arXiv preprint arXiv:2103.11595 (Accepted by DAC) (2021).

[16] HUANG, C., ZHANG, F., NEWMAN, M., CALI, J., GAO, X., TIAN, Z., WU, J., XU, H., YU, H., YUAN, B., ET AL. Classical simulation of quantum supremacy circuits. arXiv preprint arXiv:2005.06787 (2020).

[17] JI, R., WU, B., YING, M., SUN, X., AND YANG, G. Quantum supremacy circuit simulation on sunway taihulight. IEEE Transactions on Parallel and Distributed Systems 31, 4 (2019), 805–816.

[18] MARKOV, I. L., AND SHI, Y. Simulating quantum computation by contracting tensor networks. SIAM Journal on Computing 38, 3 (2008), 963–981.

[19] MILLER, D. M., AND THORONTON, M. A. Qmdd: A decision diagram structure for reversible and quantum circuits. In 36th International Symposium on Multiple-Valued Logic (ISMVL’06) (2006), IEEE, pp. 30–33.

[20] MOLITOR, P., AND MOHNKE, J. Equivalence checking of digital circuits: fundamentals, principles, methods. Springer Science & Business Media, 2007.

[21] MOLL, N., ET AL. Quantum optimization using variational algorithms on near-term quantum devices. Quantum Science and Technology 3, 3 (2018), 030503.

[22] NIELSEN, M. A., AND CHUANG, I. Quantum computation and quantum information, 2002.

[23] NIEMANN, P., WILLE, R., MILLER, D. M., THORONTON, M. A., AND DRECHSLER, R. Qmdd: Efficient function computation and representation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35, 1 (2015), 86–99.

[24] PEDNAULT, E., GUNNELS, J. A., NANNICINI, G., HORESH, L., MAGERLEIN, T., SOLOMONIK, E., DRAEGER, E. W., HOLLAND, E. T., AND WINSIEFF, R. Breaking the 49-qubit barrier in the simulation of quantum circuits. arXiv preprint arXiv:1710.05867 (2017).

[25] BALAHINE, C., MILLER, A., GANANIL, M., ZALAMAIN, A., FONTAINE, B., ZOU, Y., HIDARY, J., VIDAL, G., AND LEICHENHAUSEN, S. Tensor network: A library for physics and machine learning. arXiv preprint arXiv:1905.01330 (2019).

[26] VIAMONTES, G. F., MARKOV, I. L., AND HAYES, J. P. Improving gate-level simulation of quantum circuits. Quantum Information Processing 2, 5 (2003), 347–380.

[27] WILLE, R., HILLMICH, S., AND BURGHOHLER, L. Efficient and correct compilation of quantum circuits. In IEEE International Symposium on Circuits and Systems (2020).

[28] ZULEHNER, A., PALER, A., AND WILLE, R. An efficient methodology for mapping quantum circuits to the ibm qx architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 7 (2018), 1226–1236.

[29] ZULEHNER, A., AND WILLE, R. Advanced simulation of quantum computations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38, 5 (2018), 848–859.

APPENDIX

PROOFS AND EMPIRICAL RESULTS

Detailed Proofs

Lemma 6 (Lemma 2). For any tensor φ which is not normal, there exists a unique normal tensor φ∗ such that φ = p · φ∗, where p is a nonzero complex number.

Proof. Since φ is not normal, we have φ ≠ 0. Let p = φ(ǎ∗) where ǎ∗ is the pivot of φ. Then φ∗ := ½ · p is a normal tensor which satisfies the condition. Furthermore, suppose there is another normal tensor φ′ such that φ′ = p′ · φ′′ for some complex number p′. Then we have φ = p · φ′ = p′ · φ″. Obviously, we have |p| = |p′| and, by definition, φ* and φ′ are also the same pivot ǎ∗ with φ. It then follows that φ∗(ǎ∗) = φ″(ǎ∗) = 1. Thus p = p′, and φ∗ = φ′.

Lemma 7. Every terminal node of a normal TDD F has value 0 or 1. Moreover, let v be a non-terminal node of F with Φ(v) ≠ 0, and w₀ and w₁ the weights on its low- and high-edge. Then we have either w₀ = 1 or w₁ = 1.

Proof. The terminal case is clear by definition. Suppose the index set of F = {x₁, ..., xₙ} and x₁ < ... < xₙ. For a non-terminal node v, let φ, φ₁, and φ₀ denote Φ(v), Φ(low(v)), and Φ(high(v)), respectively. Then φ₁ = φ₀, π₁ φ₀ w₁ + x₁ φ₁ by Eq. 1 where x₁ = index(v). Note that φ is a tensor over {x₁, ..., xₙ} and both φ₁ and φ₀ can be regarded as tensors over {x₁, ..., xₙ}.

Let ǎ∗ be the pivot of φ. Suppose ǎ∗ = 0b∗ for some b∗ ∈ {0, 1}ⁿ−₁; that is, ǎ∗ takes value 0 at index x₁. Then |φ(ǎ∗)| = w₀ · φ₀(b∗) ≥ 1 from the fact that φ₀ is normal. On the other hand, let ć be the pivot of φ₁. Then from φ₀(ć) = w₀ · φ₀(ć) = w₀ and the fact that φ is normal, we have |w₀| ≤ 1. Thus |w₁| = 1 and |φ₁(b∗)| = 1. Now for any b ∈ {0, 1}ⁿ−₁ which is less than b∗ in the lexicographic order, we have |φ₀(b)| = |φ₀(b)| < |φ₀(ǎ∗)| = 1, as 0b is less than 0b∗ = ǎ∗. Thus by definition, ć is actually the pivot of φ₁. So φ₁(b*) = 1 and hence w₁ = 1.

The case when ǎ∗ takes value 1 at index x₁ is analogous.

Theorem 6 (Theorem 3). Let I = {x₁, x₂, ..., xₙ} be a set of indices and ≤ a linear order on it. For any tensor φ with index set I, there exists a ≺-ordered normal TDD F such that Φ(F) = φ.

Proof. We prove the result by induction on the cardinality of I. If |I| = 0, the tensor is a constant and the conclusion clearly holds after possible application of NR1. Suppose the statement
holds for tensors with up to \( n \) indices. We show it is also true for tensors with \( n + 1 \) indices. Let \( I = \{x_1, \ldots, x_{n+1}\} \) be the index set and, without loss of generalisation, assume \( x_1 < x_2 < \ldots < x_{n+1} \). Given an arbitrary tensor \( \phi \) over \( I \), by the Boolean-Schonhage expansion, we know
\[
\phi = \Phi_1 + \cdot \phi_0 + x_1 \cdot \phi_1,
\]
where \( \phi_c := \phi|_{x_c=0} \) for \( c \in \{0, 1\} \). Since \( \phi_c \) is a tensor over \( n \) indices, by induction hypothesis, there is a \( \prec \)-ordered normal TDD \( F_c \) such that \( \phi_c = \Phi(F_c) \), where \( \prec \) is the restriction of \( \prec \) on \( I \setminus \{x_1\} \). Let \( r_c \) be the root node and \( w_c := w_{F_c} \) the weight of \( F_c \). Then we have \( \phi_c = \Phi(F_c) = w_i \cdot \Phi(r_c) \). Next, we introduce a new root node \( v \) with weight 1 on its incoming edge. Set \( \text{low}(v) \) and \( \text{high}(v) \) to be \( r_0 \) and \( r_1 \) respectively. Furthermore, set the weights on the low- and high-edges of \( v \) to be \( w_0 \) and \( w_1 \), respectively. The constructed TDD, denoted by \( F \), is \( \prec \)-ordered and, after applying the normalisation rule \( \text{NR2} \) on \( v \), normal. By Eq.\[7\] we have \( \Phi(F) = \phi \). \( \square \)

**Lemma 8 (Lemma\[4\]).** Suppose \( \mathcal{F} \) is a reduced TDD of a non-constant tensor \( \phi \) over index set \( I \). Then every non-terminal node of \( \mathcal{F} \) is labelled with an index that is essential to \( \phi \).

**Proof.** Suppose \( v \) is a non-terminal node of \( \mathcal{F} \) which is labelled with a non-essential index \( x \). Let \( \phi' = \Phi(v) \). Then \( \phi'|_{x=0} = \phi'|_{x=1} \). From Eq.\[7\] we have \( \phi'|_{x=0} = w_0 \cdot \Phi(\text{low}(v)) \) and \( \phi'|_{x=1} = w_1 \cdot \Phi(\text{high}(v)) \), where \( w_0 \) and \( w_1 \) are the weights on the low- and high-edges of \( v \), respectively. It follows by Lemma\[2\] that \( \Phi(\text{low}(v)) = \Phi(\text{high}(v)) \) and \( w_0 = w_1 \) since they are both normal. Note that \( \text{low}(v) \) and \( \text{high}(v) \) may be identical. From Lemma\[7\] we have \( w_0 = w_1 = 1 \) and thus \( \Phi(v) = \Phi(\text{low}(v)) + x \cdot \Phi(\text{high}(v)) = \Phi(\text{low}(v)) \). This shows that we have two nodes, viz. \( v \) and \( \text{low}(v) \), representing the same tensor, which contradicts the assumption that \( \mathcal{F} \) is reduced. \( \square \)

**Theorem 7 (canonicity, Theorem\[5\]).** Let \( I \) be an index set and \( \prec \) a linear order on \( I \). Suppose \( \mathcal{F} \) and \( \mathcal{G} \) are two \( \prec \)-ordered, reduced TDDs over \( I \) with \( \Phi(\mathcal{F}) = \Phi(\mathcal{G}) \). Then \( \mathcal{F} \approx \mathcal{G} \).

**Proof.** We prove this by induction on the cardinality of \( I \). First, reduced TDDs of any constant tensor are clearly unique. In particular, from 1) and 2) of Definition\[5\], the 0 tensor is represented by the reduced TDD with weight 0 which has a unique node, viz. terminal 1.

Suppose the statement holds for any tensor with at most \( n \) indices. We prove it also holds for tensors with \( n + 1 \) indices. From \( \Phi(\mathcal{F}) = \Phi(\mathcal{G}) \), we have by Lemma\[3\] that \( \Phi(r_\mathcal{F}) = \Phi(r_\mathcal{G}) \) and \( w_\mathcal{F} = w_\mathcal{G} \). In addition, by Lemma\[4\] \( r_\mathcal{F} \) and \( r_\mathcal{G} \) are labeled with essential indices. They must be the same as, otherwise, the smaller one in the order \( \prec \) is not essential for either \( \mathcal{F} \) or \( \mathcal{G} \). Let \( x \) be this variable. By Lemma\[5\] we have
\[
\Phi(\mathcal{F}) = \Phi(\mathcal{F}|_{x=0}) + x \cdot \Phi(\mathcal{F}|_{x=1})
\]
\[
\Phi(\mathcal{G}) = \Phi(\mathcal{G}|_{x=0}) + x \cdot \Phi(\mathcal{G}|_{x=1}).
\]
Since \( \Phi(\mathcal{F}) = \Phi(\mathcal{G}) \), it holds that \( \Phi(\mathcal{F}|_{x=c}) = \Phi(\mathcal{G}|_{x=c}) \) for \( c \in \{0, 1\} \). By the induction hypothesis, we have \( \mathcal{F}_c \approx \mathcal{G}_c \). This, together with the fact that \( \text{index}(r_\mathcal{F}) = \text{index}(r_\mathcal{G}) \), implies that \( \mathcal{F} \approx \mathcal{G} \). \( \square \)

**Theorem 8 (Theorem\[4\]).** A normal TDD is reduced if and only if no reduction rule is applicable.

**Proof.** Clearly, a normal TDD \( \mathcal{F} \) is not reduced if at least one reduction rule is applicable as, otherwise, we shall have either a node representing tensor 0 or two nodes representing the same tensor.

On the other hand, suppose no reduction rule is applicable to \( \mathcal{F} \). We show by induction on the depth of \( \mathcal{F} \) that \( \mathcal{F} \) is reduced. First, from the fact that RR1 and RR2 are not applicable, \( \mathcal{F} \) must have a unique terminal node with value 1, and all edges weighted 0 have been redirected to it.

Assume there exist non-terminal nodes which represent tensor 0 and \( v \) is such a node with the \( \prec \)-largest label. By our assumption and that \( \text{label}(v) \prec \text{label}(	ext{low}(v)) \) and \( \text{label}(v) \prec \text{label}(	ext{high}(v)) \), we have \( \Phi(\text{low}(v)) \neq 0 \) and \( \Phi(\text{high}(v)) \neq 0 \). Now, as \( \Phi(v) = 0 \), the weights on the low- and high-edges of \( v \) must both be 0, which however activates either RR2 or RR3 and thus a contradiction with our assumption.

Suppose there are two non-terminal nodes \( v \) and \( v' \) \( (\Phi(v) = \Phi(v')) \). Let \( F_v \) and \( F_{v'} \) be the sub-TDDs of \( \mathcal{F} \) rooted at \( v \) and \( v' \) respectively (but set their weights to be 1). Note that no reduction rule is applicable to either \( F_v \) or \( F_{v'} \), since otherwise it is also applicable to \( \mathcal{F} \) . Then by induction hypothesis, they are both reduced. Furthermore, we have \( \Phi(F_v) = \Phi(v) = \Phi(v') = \Phi(F_{v'}), \) and from Theorem\[3\] \( F_v \approx F_{v'} \). Then \( \text{index}(v) = \text{index}(v') \) and \( w_0 = w_0' \), where \( w_0 \) and \( w_0' \) are the weights on the low-edges of \( v \) and \( v' \), respectively. Furthermore, it follows from Eq.\[6\] that \( \Phi(\text{low}(v)) = \Phi(\text{low}(v')) \). By induction hypothesis, we have \( \text{low}(v) = \text{low}(v') \). Similarly, we can prove that \( \text{high}(v) = \text{high}(v') \) as well. That is, RR4 is applicable to \( v \) and \( v' \) and thus also a contradiction with our assumption.

In summary, we have shown that \( \mathcal{F} \) is reduced. \( \square \)

**Empirical Results**
| Name        | Qubit num | Gate num | Time | num. max | num. final |
|-------------|-----------|----------|------|----------|-----------|
| grayscale6.47 | 6         | 5        | 0.03 | 12       | 12        |
| ex-1_166    | 3         | 19       | 0.03 | 10       | 9         |
| 4mod5_v0.20 | 5         | 20       | 0.03 | 28       | 22        |
| rd32-v0.66  | 4         | 34       | 0.04 | 14       | 9         |
| deco24-v0.38| 4         | 51       | 0.04 | 18       | 16        |
| 4gt13.92    | 5         | 66       | 0.06 | 26       | 26        |
| 4mod5-bdd287| 7         | 70       | 0.07 | 111      | 74        |
| alu_v0.26   | 5         | 84       | 0.06 | 46       | 23        |
| 4gt5.76     | 5         | 91       | 0.07 | 50       | 20        |
| 4gt5.77     | 5         | 131      | 0.11 | 46       | 27        |
| deco24-v3.45| 5         | 150      | 0.13 | 46       | 17        |
| cnt3_5_179  | 16        | 175      | 0.19 | 55       | 48        |
| 0410184_169 | 14        | 211      | 0.20 | 63       | 39        |
| sys6-v0.111 | 10        | 215      | 0.38 | 473      | 247       |
| 4gt4-v0.72  | 6         | 258      | 0.20 | 98       | 31        |
| sim6_315    | 14        | 270      | 1.21 | 5606     | 1502      |
| simy_146    | 12        | 328      | 0.49 | 523      | 229       |
| mod8_10_178 | 6         | 342      | 0.72 | 18       | 10        |
| rd8_142     | 15        | 343      | 3.56 | 6922     | 3588      |
| alu_v2.31   | 5         | 451      | 0.18 | 44       | 18        |
| cnt3_5_180  | 16        | 485      | 0.38 | 164      | 48        |
| rd53_133    | 7         | 580      | 0.29 | 78       | 26        |
| majority_239| 7         | 612      | 0.30 | 87       | 17        |
| sf_276      | 6         | 778      | 0.37 | 59       | 19        |
| conf_3_16   | 9         | 954      | 0.60 | 214      | 37        |
| cm4a_207    | 14        | 1776     | 1.21 | 211      | 81        |
| hwbv_56_76  | 5         | 6723     | 8.57 | 242      | 88        |
| sqn_258     | 10        | 10223    | 12.28| 438      | 105       |
| sum(MO)     |           |          |      | 30.66    | -         |
| sum         |           |          |      | 31.23    | 15758     |
| bv_10       | 10        | 29       | 0.04 | 20       | 20        |
| bv_20       | 20        | 59       | 0.08 | 40       | 40        |
| bv_30       | 30        | 89       | 0.13 | 60       | 70        |
| bv_40       | 40        | 119      | 0.19 | 80       | 122       |
| bv_50       | 50        | 149      | 0.28 | 100      | 96        |
| bv_60       | 60        | 179      | 0.40 | 120      | 112       |
| bv_70       | 70        | 209      | 0.50 | 140      | 122       |
| bv_80       | 80        | 239      | 0.58 | 160      | 148       |
| bv_90       | 90        | 269      | 0.71 | 180      | 168       |
| bv_100      | 100       | 299      | 0.84 | 200      | 196       |
| qy_n2_d5    | 2         | 50       | 0.12 | 6        | 6         |
| qy_n3_d5    | 3         | 50       | 0.09 | 22       | 22        |
| qy_n4_d5    | 4         | 100      | 0.29 | 86       | 86        |
| qy_n5_d5    | 5         | 100      | 0.49 | 342      | 342       |
| qy_n6_d5    | 6         | 150      | 1.36 | 1366     | 1366      |
| qy_n7_d5    | 7         | 150      | 1.66 | 16462    | 16462     |
| qy_n8_d5    | 8         | 200      | 2.27 | 21846    | 21846     |
| qy_n9_d5    | 9         | 200      | 3.52 | 78387    | 78382     |
| qft_5       | 5         | 15       | 0.04 | 32       | 32        |
| qft_6       | 6         | 21       | 0.08 | 64       | 64        |
| qft_7       | 7         | 28       | 0.06 | 128      | 128       |
| qft_8       | 8         | 36       | 0.11 | 256      | 256       |
| qft_9       | 9         | 45       | 0.16 | 512      | 512       |
| qft_10      | 10        | 55       | 0.32 | 1024     | 1024      |
| qft_11      | 11        | 66       | 0.65 | 2048     | 2048      |
| qft_12      | 12        | 78       | 1.17 | 4096     | 4096      |
| qft_13      | 13        | 91       | 2.39 | 8192     | 8192      |
| qft_14      | 14        | 105      | 5.62 | 16384    | 16384     |
| qft_15      | 15        | 120      | 12.06| 32768    | 32768     |
| qft_16      | 16        | 136      | 29.10| 65366    | 65366     |
| qft_17      | 17        | 153      | 74.54| 131072   | 131072    |
| qft_18      | 18        | 171      | 184.63| 261244   | 261244    |
| qft_19      | 19        | 190      | 439.49| 524288   | 524288    |
| qft_20      | 20        | 210      | 1067.45| 1048576  | 1048576   |
| qft_21      | 21        | 231      | 2960.98| 2097152  | 2097152   |
| qft_22      | 22        | 253      | 3523.67| 87382    | 87382     |
| sum(TO)     |           |          |      | 5104.69  | 4240260   |

| Name        | Qubit num | Gate num | Time | num. max | num. final |
|-------------|-----------|----------|------|----------|-----------|
| TDD No Part. |           |          |      | 0.01     | 22        |
| TDD Part. I |           |          |      | 0.01     | 22        |
| TDD Part. II|           |          |      | 0.01     | 22        |
| TN          |           |          |      | 0.01     | 22        |

* TN represents the Google Tensor network package. MO and TO represent, respectively, output of memory time and output of memory time. * For TDD and QMDD, we list the time (seconds), max number of nodes and final number of nodes in the construction process. For TDD with the two partition schemes, we remove the final number of nodes, as they are identical to that of TDD with no partition.