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 design 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 [3] 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 qubits will very likely appear in the next 5-10 years. Several quantum algorithms have been proposed to further demonstrate practical usability of these NISQ (noise intermediate-scale quantum) devices [14], [20], [19]. 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. 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 the Quantum Information Decision Diagram (QuIDD) [26] and the Quantum Multiple-Valued Decision Diagrams (QMDD) [24]. The QuIDD is a variant of the Algebraic Decision Diagram (ADD) [4] by restricting values to complex numbers, which are indexed by integers, and interleaving row and column variables in the variable ordering. In contrast, the 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 [24] 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 [34] and equivalence checking [10], [9] of quantum circuits as well as verifying the correctness of quantum circuits compilation [28]. On the other hand, using QMDDs to represent quantum circuits has the following two drawbacks. First, any gate must be extended to act on the whole set of qubits when constructing its QMDD representation. To be specific, in an n-qubit circuit we have to tensor each k-qubit gate with the identity matrices on the remaining $n - k$ qubits [24]. Note that in practical quantum circuits, basic gates are usually 1- or 2-qubit ones. These operations sometimes could be clumsy and time-consuming, and the obtained QMDD can be less compact. For example, while a CX gate is represented as a QMDD with only 4 nodes, in a quantum circuit with 20 qubits, it requires a 40-node QMDD to represent the CX gate between the first and the last qubits. Second, the computation of QMDD representation of a quantum circuit has to be done following the gate order, typically from left to right, which prevents us from exploiting the special topological structure presented in many practical circuits.

Tensor network provides a more flexible way to represent quantum circuits, from which the functionality of the circuit can be computed in essentially any order, while the QMDD simulation of quantum circuits corresponds to contracting the tensors (representing individual quantum gates in the circuit) sequentially from left to right in their original order. Indeed, 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 NP-hard to find an optimal contraction order in general, one may exploit heuristics like circuit partition, which was recently demonstrated as very useful for simulating quantum circuits in, e.g., [25]. The key idea is, as depicted

Xin Hong, Sanjiang Li, Yuan Feng and Mingsheng Ying are with Centre for Quantum Software and Information (QSI), Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia.
E-mail: {sanjiang.li, yuan.feng, mingsheng.ying}@uts.edu.au
Xiangzhen Zhou is with Centre for Quantum Software and Information (QSI), University of Technology Sydney, NSW 2007, Australia and State Key Lab of Millimeter Waves, Southeast University, Nanjing 211189, China.
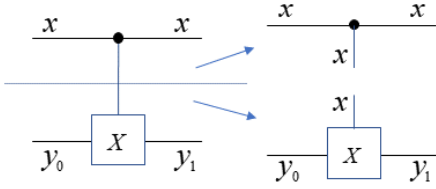
Fig. 1. Example of cutting CX gate.

Fig. 2. The matrix representations of the H, T, and CX gate.

in Fig. 1, 'cutting' the tensor representing a CX gate into two tensors, one called COPY and the other XOR [5]. Here, COPY takes value 1 if the values of the three indices are identical and 0 otherwise; XOR takes value 1 if the values a, b, c of the three indices satisfy $a \oplus b \oplus c = 0$ and takes value 0 otherwise. Obviously, this 'cut' operation is the reverse operation of tensor contraction. Note that it seems difficult, if not impossible, to incorporate the circuit partition technique in the QMDD or QuIDD data structure of quantum circuits.

The aim of this paper is to define a novel decision diagram, called Tensor Decision Diagram (TDD for short), which can potentially be used together with QMDD in a complementary manner so that the drawbacks of QMDD discussed above could be partially remedied. TDD is inspired by the success of tensor networks in classical simulation of quantum circuits in the last few years. By observing that quantum circuits are a special class of tensor networks, Pednault et al. [25] 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. [6], [17], [11], [12], [16]). To further extend this flexibility in a more principled way, we propose TDD as a new data structure for tensor networks. Furthermore, tensors are essentially multidimensional linear maps with complex values, which also enjoy Boole-Shannon style expansions. This observation leads us to design TDD as decision diagram.

TDDs have several important features that warrant their applicability. Analogous to the well-known decision diagram ROBDD for Boolean functions [8], 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 thus reduced form of TDD provides, up to variable ordering, a unique representation for the functionality of a quantum circuit. A reduced TDD $\mathcal{F}$ is a rooted direct acyclic graph which has one terminal node labeled with 1 and zero or many non-terminal nodes. The root node has a unique incoming edge which bears the 'weight' of the tensor $\mathcal{F}$ represents, whose magnitude equals to the maximum norm of the tensor. Each non-terminal node has two outgoing edges, called its low- and high-edges, one of which has weight 1, the other has weight w with $|w| \leq 1$. We present an efficient algorithm 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
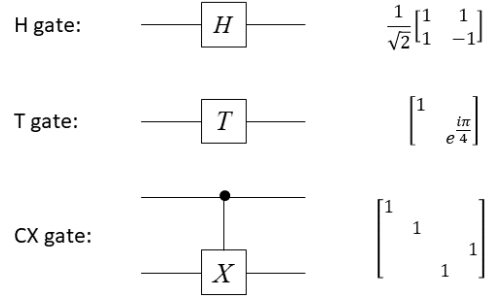
contraction can be implemented efficiently.

As QMDD, TDD provides a universal, compact and canonical representation for quantum circuits, which is vital in various design automation tasks. Generating the TDD representation of a quantum circuit can, for example, verify its correctness. A byproduct of our evaluation indeed shows that the widely used 'qft_10' and 'qft_16' circuits[1] are incorrect as their TDD representations are the same as the identity unitary.

In the remainder of this paper, after a brief review of quantum circuits and QMDD in Sec. II 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.

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

In classical computation, data are represented by a string of bits. When sending through a classical circuit, the state of the input data will be transformed by a sequence of logical gates. In quantum computing, the counterpart of bit is called *qubit*. The state of a qubit is often represented in Dirac notation

$$|\varphi\rangle := \alpha_0 |0\rangle + \alpha_1 |1\rangle, \tag{1}$$

where α_0 and α_1 are complex numbers, called the amplitudes of $|\varphi\rangle$, and $|\alpha_0|^2 + |\alpha_1|^2 = 1$. We also use the vector $[\alpha_0, \alpha_1]^{\mathsf{T}}$ to represent a single-qubit state. In general, an n-qubit quantum state is represented as a 2^n-dimensional complex vector $[\alpha_0, \alpha_1, \ldots, \alpha_{2^n-1}]^{\mathsf{T}}$.

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. 2

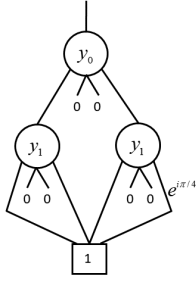[1] https://github.com/iic-jku/ibm_qx_mapping/tree/master/examples

Fig. 3. The QMDD representation of the controlled-T gate, where the weight of an edge is omitted if it is 1.

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 resulted from applying a Hadamard gate to an input state $[\alpha_0, \alpha_1]^{\mathsf{T}}$ 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 are applied to the input state in a sequential manner. Clearly, a quantum circuit describes a quantum functionality and is also represented as a $2^n \times 2^n$-dimensional unitary transformation matrix.

B. Quantum Multi-value Decision Diagram

Quantum Multi-value Decision Diagram (QMDD) [21] 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 thus 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. 3 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 $\phi : \{0, 1\}^I \to \mathbb{C}$, 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 $\phi_{\vec{x}}(\vec{a})$ 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 2×2 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

$$\phi_{\vec{x}, \vec{y}}(\vec{a}, \vec{b}) = \sum_{\vec{c} \in \{0,1\}^{\vec{z}}} \gamma_{\vec{x}, \vec{z}}(\vec{a}, \vec{c}) \cdot \xi_{\vec{y}, \vec{z}}(\vec{b}, \vec{c}). \tag{2}$$

Another useful tensor operation is *slicing*, which corresponds to the cofactor operation of Boolean functions. Let ϕ be a tensor with index set $I = \{x, x_1, \ldots, x_n\}$. The slicing of ϕ 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

$$\phi|_{x=c}(\vec{a}) := \phi(c, \vec{a}) \tag{3}$$

for any $\vec{a} \in \{0, 1\}^n$. We call $\phi|_{x=0}$ and $\phi|_{x=1}$ the negative and positive slicing of ϕ with respect to x, respectively. We say an index $x \in I$ is *essential* for ϕ 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 [5] for more detailed introduction.

B. Quantum circuits as tensor networks

The quantum state of a qubit x with vector representation $[\alpha_0, \alpha_1]^{\mathsf{T}}$ can be described as a rank 1 tensor ϕ_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 ϕ_{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
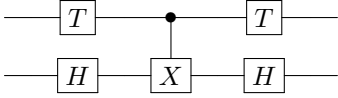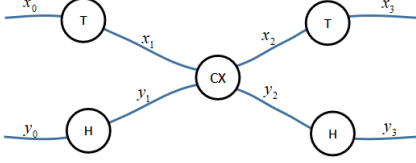
Fig. 4. A quantum circuit.

Fig. 5. A tensor network which is equivalent to the circuit shown in Fig. 4.

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 $2n$ 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 $2n$, 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. 4. Regarding each gate as a tensor (cf. Fig. 2), Fig. 5 shows the tensor network representation of the circuit. By contracting the tensor network, we obtain the tensor representation of the circuit*

$$\phi_{x_0 x_3 y_0 y_3}(a_0 a_3 b_0 b_3) =$$
$$\sum_{a_1,a_2,b_1,b_2=0}^{1} T(a_0 a_1) H(b_0 b_1) CX(a_1 b_1 a_2 b_2) T(a_2 a_3) H(b_2 b_3). \quad (4)$$

It is straightforward to check that this tensor indeed gives the functionality of the circuit presented in Fig. 4. For example, $\phi_{x_0 x_3 y_0 y_3}(1111) = -i$ corresponds to the fact that the circuit maps $|11\rangle$ to $-i|11\rangle$.

Given a tensor $\phi_{\vec{x}}$ and $x_i, x_j \in \vec{x}$, if $\phi_{\vec{x}}(\vec{a}) = 0$ 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 tenor for Z gate
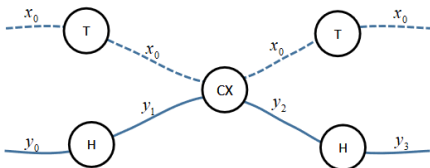

Fig. 6. A tensor network with hyper-edge (denoted by the dotted line), which is also equivalent to the circuit shown in Fig. 4.

can be written as ϕ_{xx} with $\phi_{xx}(0) = 1$ and $\phi_{xx}(1) = -1$. Similarly, CX gate can be represented as a tensor $\phi_{xxy_1y_2}$ with $\phi_{xxy_1y_2}(abc) = a \cdot (b \oplus c) + \overline{a} \cdot \overline{b \oplus c}$, where $\overline{a}$, for example, is the complement of a. In [25], edges formed by identical indices are called *hyper-edges*.

Example 3. *For the tensor network shown in Fig. 5, 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. 6, where the dotted line is a hyper-edge and the corresponding tensor becomes $\phi_{x_0 x_0 y_0 y_3}$.*

IV. TENSOR DECISION DIAGRAM

To fully exploit the benefit of tensor representation of quantum circuits and the circuit partition technique, a suitable data structure for tensors is desired. In this section, we introduce such a data structure — Tensor Decision Diagram (TDD).

A. Basic Definition

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

Lemma 1. *Let ϕ be a tensor with indices in I. For each $x \in I$,*

$$\phi = \overline{x} \cdot \phi|_{x=0} + x \cdot \phi|_{x=1}, \quad (5)$$

where $\overline{x}(c) := 1 - x(c)$ for $c \in \{0, 1\}$.

Note that in above we regard each index $x \in I$ as the identity tensor with only one index x, which maps 0 to 0 and 1 to 1.

Recursively using the Boole-Shannon expansion, a tensor can be naturally represented with a decision diagram.

Definition 1 (TDD). *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 $\mathcal{F}$;*
- *$index : V_N \to I$ assigns each non-terminal node an index in I;*
- *$value : V_T \to \mathbb{C}$ assigns each terminal node a complex value;*
- *both low and $high$ are mappings in $V_N \to V$ 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;*
- *$w : E \to \mathbb{C}$ assigns each edge a complex weight. In particular, $w(e_r)$ is called the weight of $\mathcal{F}$, and denoted $w_{\mathcal{F}}$.*

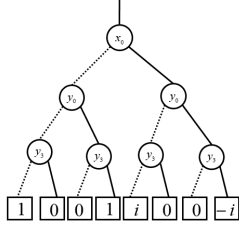*A TDD is called* trivial *if its root node is also a terminal node.*

Fig. 7. A TDD representation of the tensor in Example 3.

For convenience, we often call a terminal node with value c a terminal c node or simply terminal c if it is unique. The size of a TDD $\mathcal{F}$, written $size(\mathcal{F})$, is the number of non-terminal nodes of $\mathcal{F}$. As each non-terminal node has two outgoing edges, there are altogether $1 + 2 \times size(\mathcal{F})$ edges, including e_r, in $\mathcal{F}$.

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

Example 4. *Fig. 7 gives the TDD obtained by directly applying the Boole-Shannon expansion to the tensor $\phi_{x_0 x_0 y_0 y_3}$ 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 ϕ under the evaluation specified by the path. For example, the terminal node with value i corresponds to the value of ϕ 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 $\mathcal{F}$ naturally corresponds to a tensor $\Phi(v)$. If v is a terminal node, then $\Phi(v) := value(v)$ is a rank 0 tensor, i.e., a constant; if v is a non-terminal node, then

$$\Phi(v) := w_0 \cdot \overline{x}_v \cdot \Phi(low(v)) + w_1 \cdot x_v \cdot \Phi(high(v)), \quad (6)$$

where $x_v = index(v)$, 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

$$\Phi(v)|_{x_v = c} = w_c \cdot \Phi(v_c), \quad (7)$$

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

Finally, the tensor represented by $\mathcal{F}$ itself is defined to be

$$\Phi(\mathcal{F}) := w_{\mathcal{F}} \cdot \Phi(r_{\mathcal{F}}). \quad (8)$$

Recall here that $r_{\mathcal{F}}$ and $w_{\mathcal{F}}$ are the root node and the weight of $\mathcal{F}$, respectively.

An efficient manipulation of general TDDs seems impossible. Following [8], we restrict our discussion to ordered TDDs.

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

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

Like ROBDDs, the size of the TDD representation strongly relies on the selected variable order. For example, the tensor $\phi = (x_1 \cdot x_2) + (x_3 \cdot x_4) + (x_5 \cdot x_6)$ can be represented as a TDD with 6 non-terminal nodes under the order $\prec_1 := (x_1, x_2, x_3, x_4, x_5, x_6)$, but its TDD representation under $\prec_2 := (x_1, x_3, x_5, x_2, x_4, x_6)$ requires at least $2 \times (1 + 2^1 + 2^2) = 14$ internal nodes (cf. [22, Ch.3]). While finding an optimal order is NP-hard, there are efficient heuristic methods that have been devised for ROBDDs, which may be extended to TDDs.

B. Normalisation

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

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

$$\|\phi\| := \max_{\vec{a} \in \{0,1\}^I} |\phi(\vec{a})| \quad (9)$$

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

$$\vec{a}^* = \min\{\vec{a} \in \{0,1\}^I : |\phi(\vec{a})| = \|\phi\|\}. \quad (10)$$

We call $\vec{a}^*$ the pivot of ϕ. A tensor ϕ is called normal *if either $\phi = 0$ or $\phi(\vec{a}^*) = 1$. Note that tensors with maximum norm 1 are not necessarily normal.*

The following lemma shows that any tensor can be uniquely normalised.

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

Proof. Since ϕ is not normal, we have $\phi \neq 0$. Let $p = \phi(\vec{a}^*)$ where $\vec{a}^*$ is the pivot of ϕ. Then $\phi^* := \frac{1}{p} \cdot \phi$ is a normal tensor which satisfies the condition. Furthermore, suppose there is another normal tensor ϕ' such that $\phi = p' \cdot \phi'$ for some complex number p'. Then we have $\phi = p \cdot \phi^* = p' \cdot \phi'$. Obviously, we have $|p| = |p'|$ and, by definition, ϕ^* and ϕ' also share the same pivot $\vec{a}^*$ with ϕ. It then follows that $\phi^*(\vec{a}^*) = \phi'(\vec{a}^*) = 1$. Thus $p = p'$, and $\phi^* = \phi'$. $\square$

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

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

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. *Every terminal node of a normal TDD $\mathcal{F}$ has value 0 or 1. Moreover, let v be a non-terminal node of $\mathcal{F}$ with $\Phi(v) \neq 0$, and w_0 and w_1 the weights on its low- and high-edge. Then we have either $w_0 = 1$ or $w_1 = 1$.*

Proof. The terminal case is clear by definition. Suppose the index set of $\mathcal{F}$ is $\{x_1, \ldots, x_n\}$ and $x_1 \prec \ldots \prec x_n$. For a non-terminal node v, let ϕ, ϕ_l, and ϕ_h denote $\Phi(v), \Phi(low(v))$, and $\Phi(high(v))$, respectively. Then $\phi = w_0 \cdot \overline{x}_i \cdot \phi_l + w_1 \cdot x_i \cdot \phi_h$ by Eq. 6, where $x_i = index(v)$. Note that ϕ is a tensor over $\{x_i, \ldots, x_n\}$ and both ϕ_l and ϕ_h can be regarded as tensors over $\{x_{i+1}, \ldots, x_n\}$.

Let $\vec{a}^*$ be the pivot of ϕ. Suppose $\vec{a}^* = 0\vec{b}^*$ for some $\vec{b}^* \in \{0,1\}^{n-i}$; that is, $\vec{a}^*$ takes value 0 at index x_i. Then by $1 = \phi(\vec{a}^*) = w_0 \cdot \phi_l(\vec{b}^*)$, we have $|w_0| \geq 1$ from the fact that ϕ_l is normal. On the other hand, let $\vec{c}$ be the pivot of ϕ_l. Then from $\phi(0\vec{c}) = w_0 \cdot \phi_l(\vec{c}) = w_0$ and the fact that ϕ is normal, we have $|w_0| \leq 1$. Thus $|w_0| = 1$ and $|\phi_l(\vec{b}^*)| = 1$. Now for any $\vec{b} \in \{0,1\}^{n-i}$ which is less than $\vec{b}^*$ in the lexicographic order, we have $|\phi_l(\vec{b})| = |\phi(0\vec{b})| < |\phi(\vec{a}^*)| = 1$, as $0\vec{b}$ is less than $0\vec{b}^* = \vec{a}^*$. Thus by definition, $\vec{b}^*$ is actually the pivot of ϕ_l. So $\phi_l(\vec{b}^*) = 1$ and hence $w_0 = 1$.

The case when $\vec{a}^*$ takes value 1 at index x_i is analogous. $\square$

Lemma 4. *Suppose $\mathcal{F}$ and $\mathcal{G}$ are two normal TDDs such that $\Phi(\mathcal{F}) = \Phi(\mathcal{G})$. Then we have $w_\mathcal{F} = w_\mathcal{G}$ and $\Phi(r_\mathcal{F}) = \Phi(r_\mathcal{G})$.*

Proof. By Eq. 8, we have $\Phi(\mathcal{F}) = w_\mathcal{F} \cdot \Phi(r_\mathcal{F})$ and $\Phi(\mathcal{G}) = w_\mathcal{G} \cdot \Phi(r_\mathcal{G})$. Because $\Phi(r_\mathcal{F})$ and $\Phi(r_\mathcal{G})$ are normal tensors and $\Phi(\mathcal{F}) = \Phi(\mathcal{G})$, by Lemma 2, we know $w_\mathcal{F} = w_\mathcal{G}$ and $\Phi(r_\mathcal{F}) = \Phi(r_\mathcal{G})$. $\square$

For any non-normal TDD $\mathcal{F}$, 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 $value(v) \neq 1$, then set its value to 1, and change the weight w of each incoming edge of v to $value(v) \cdot w$.
NR2: Suppose v is a non-terminal node such that $\Phi(v) \neq 0$ is not normal but both $\Phi(low(v))$ and $\Phi(high(v))$ are normal. Let w_0 and w_1 be the weights on the low- and high edges of v respectively. If $\Phi(low(v)) \neq 0$ and either $\Phi(high(v)) = 0$ 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 $\mathcal{F}$ be a non-normal TDD. We first apply NR1 to every terminal node of $\mathcal{F}$ to make it normal. Furthermore, if a non-terminal node v of $\mathcal{F}$ represents a non-normal tensor but both its successors represent normal tensors. Then, it is easy to see that after applying NR2 to v, this node itself represents a normal tensor. This gives a procedure to transform $\mathcal{F}$ into a normal TDD in a bottom-up manner. Furthermore, the transformation can be done within time linear in the size of $\mathcal{F}$.

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.*

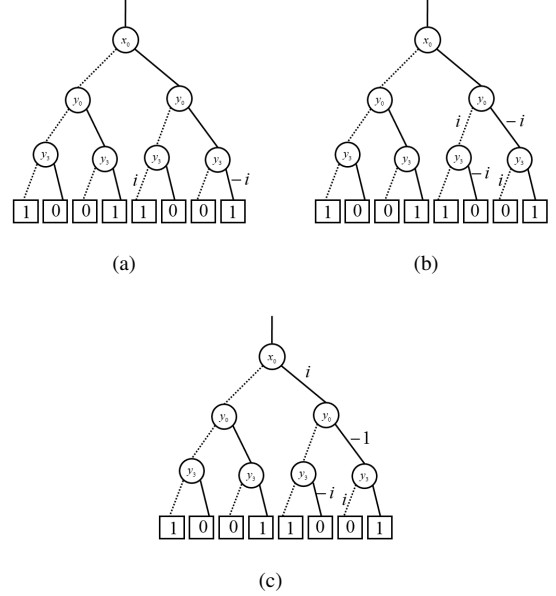*Proof.* The first part of the theorem follows from Eq. 6, and the second directly from the definitions. $\square$

Fig. 8. Normalisation of the TDD shown in Fig. 7.

Example 5. *Applying NR1 to the two terminal nodes labeled with i and $-i$ in the TDD in Fig. 7, we have the TDD as shown in Fig. 8 (a). Then, applying NR2 to the right two y_3 nodes gives the TDD in Fig. 8 (b). The normalised TDD, shown in Fig. 8 (c), is obtained by applying NR2 to the right y_0 node.*

We have seen how to transform an existing TDD into a normal one. In contrast, the following theorem provides a way to construct a normal TDD directly from a given tensor.

Theorem 2. *Let $I = \{x_1, x_2, ..., x_n\}$ be a set of indices and $\prec$ a linear order on it. For any tensor ϕ with index set I, there exists a $\prec$-ordered normal TDD $\mathcal{F}$ such that $\Phi(\mathcal{F}) = \phi$.*

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 $I = \{x_1, ..., x_{n+1}\}$ be the index set and, without loss of generalisation, assume $x_1 \prec x_2 \prec ... \prec x_{n+1}$. Given an arbitrary tensor ϕ over I, by the Boole-Shannon expansion, we know

$$\phi = \overline{x}_1 \cdot \phi_0 + x_1 \cdot \phi_1,$$

where $\phi_c := \phi|_{x_1=c}$ for $c \in \{0, 1\}$. Since ϕ_c is a tensor over n indices, by induction hypothesis, there is a $\prec'$-ordered normal TDD $\mathcal{F}_c$ such that $\phi_c = \Phi(\mathcal{F}_c)$, where $\prec'$ is the restriction of $\prec$ on $I \setminus \{x_1\}$. Let r_c be the root node and $w_c := w_{\mathcal{F}_c}$ the weight of $\mathcal{F}_c$. Then we have $\phi_c = \Phi(\mathcal{F}_c) = w_c \cdot \Phi(r_c)$. Next, we introduce a new root node v with weight 1 on its incoming edge. Set $low(v)$ and $high(v)$ 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 thus constructed TDD, denoted by $\mathcal{F}$, is $\prec$-ordered and, after applying the normalisation rule NR2 on v, normal. By Eq. 6, we have $\Phi(\mathcal{F}) = \phi$. $\square$

C. Reduction

As can been seen from Fig. 8, normal TDDs may still have redundant nodes. For example, the first and the third y_3 nodes of the normal TDD in Fig. 8(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 5. *Suppose $\mathcal{F}$ is a reduced TDD of a non-constant tensor ϕ over index set I. Then every non-terminal node of $\mathcal{F}$ is labelled with an index that is essential to ϕ.*

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, $\phi'|_{x=0} = w_0 \cdot \Phi(low(v))$ and $\phi'|_{x=1} = w_1 \cdot \Phi(high(v))$, where w_0 and w_1 are the weights on the low- and high-edges of v, respectively. It follows by Lemma 2 that $\Phi(low(v)) = \Phi(high(v))$ and $w_0 = w_1$ since they are both normal. Note that $low(v))$ and $high(v)$ may be identical. From Lemma 3, we have $w_0 = w_1 = 1$ and thus $\Phi(v) = \overline{x} \cdot \Phi(low(v)) + x \cdot \Phi(high(v)) = \Phi(low(v))$. This shows that we have two nodes, viz. v and $low(v)$, representing the same tensor, which contradicts the assumption that $\mathcal{F}$ is reduced. $\square$

The following definition of sub-TDDs is useful in our later discussion. Recall that we assume $x \prec index(v)$ 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 $x \preceq index(r_{\mathcal{F}})$. We define sub-TDDs $\mathcal{F}_{x=0}$ and $\mathcal{F}_{x=1}$ of $\mathcal{F}$ as follows.*
1) *If $x \prec index(r_{\mathcal{F}})$, then $\mathcal{F}_{x=0} = \mathcal{F}_{x=1} = \mathcal{F}$;*
2) *If $x = index(r_{\mathcal{F}})$, $\mathcal{F}_{x=0}$ is defined as the TDD rooted at $low(r_{\mathcal{F}})$ with weight $w_{\mathcal{F}} \cdot w(r_{\mathcal{F}}, low(r_{\mathcal{F}}))$, 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 6. *Suppose $\mathcal{F}$ is a reduced TDD on I, $x \in I$ and $x \preceq index(r_{\mathcal{F}})$. Then we have*

$$\Phi(\mathcal{F}) = \overline{x} \cdot \Phi(\mathcal{F}_{x=0}) + x \cdot \Phi(\mathcal{F}_{x=1}). \tag{11}$$

Now we are ready to prove the canonicity of reduced TDDs. Two TDDs $\mathcal{F}$ and $\mathcal{G}$ are said to be isomorphic, denoted $\mathcal{F} \approx \mathcal{G}$, 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 $\mathcal{F} \approx \mathcal{G}$.*

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 4 that $\Phi(r_{\mathcal{F}}) = \Phi(r_{\mathcal{G}})$ and $w_{\mathcal{F}} = w_{\mathcal{G}}$. In addition, by Lemma 5, $r_{\mathcal{F}}$ 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 6, we have

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

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 $index(r_{\mathcal{F}}) = index(r_{\mathcal{G}})$, implies that $\mathcal{F} \approx \mathcal{G}$. $\square$

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.

RR2: Redirect all weight-0 edges to the terminal. If these include the incoming edge of the root node, then the terminal becomes the new root. 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.*

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 $label(v) \prec label(low(v))$ and
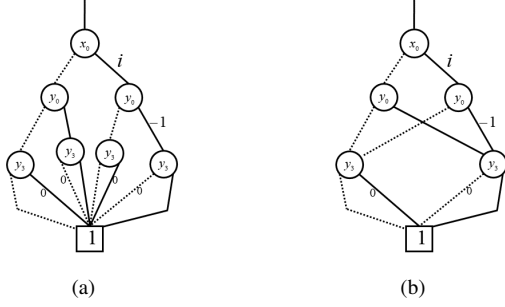
Fig. 9. Reduction of the normalised TDD shown in Fig. 8(c)

$label(v) \prec label(high(v))$, we have $\Phi(low(v)) \neq 0$ and $\Phi(high(v)) \neq 0$. 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' with $\Phi(v) = \Phi(v')$. Let $\mathcal{F}_v$ and $\mathcal{F}_{v'}$ 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 $\mathcal{F}_v$ or $F_{v'}$, since otherwise it is also applicable to $\mathcal{F}$. Then by induction hypothesis, they are both reduced. Furthermore, we have $\Phi(\mathcal{F}_v) = \Phi(v) = \Phi(v') = \Phi(\mathcal{F}_{v'})$, and from Theorem 3, $\mathcal{F}_v \approx \mathcal{F}_{v'}$. Then $index(v) = index(v')$ 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 $\Phi(low(v)) = \Phi(low(v'))$. By induction hypothesis, we have $low(v) = low(v')$. Similarly, we can prove that $high(v) = high(v')$ 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. □

The following theorem guarantees that the reduced TDD of a tensor can be obtained by applying the reduction rules.

Theorem 5. *Let $\mathcal{F}$ be a normal TDD representing tensor ϕ. Applying a reduction rule to $\mathcal{F}$ does not change the tensor it represents. Moreover, the reduced TDD of ϕ can be obtained from $\mathcal{F}$ 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 $\mathcal{F}$ 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 $\phi = \Phi(\mathcal{F})$. As reduced TDDs are unique (see Theorem 3), this gives the reduced TDD of ϕ. □

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

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

Example 6. *Consider the normalised TDD shown in Fig. 8(c). Applying RR1 to merge all terminal 1 nodes and delete all*

terminal 0 nodes gives the TDD shown in Fig. 9 (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. 9 (b), which provides a compact representation for the circuit in Fig. 5.

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 [7], 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 ϕ over a linearly ordered index set I.
Output: The reduced TDD of ϕ.
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 ϕ
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 ϕ_{xxy}, then the two successors of the node representing ϕ_{xxy} will be ϕ_{00y} and ϕ_{11y}. In other words, we construct the TDD as if it is the tensor ϕ_{xy}. When tensor operations are concerned, however, both x indices will be involved.

Example 7. *Consider the CX gate shown in Fig. 6, which is represented by a tensor $\phi_{x_0 x_0 y_1 y_2}$. The reduced TDD of $\phi_{x_0 x_0 y_1 y_2}$ is shown in Fig.10 (b), where the index x_0 only appears once with the two successors representing the tensor $\phi_{00 y_1 y_2}$ and $\phi_{11 y_1 y_2}$.*
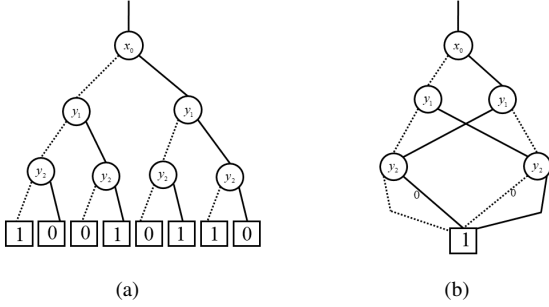
Fig. 10. Two TDDs of the CX gate with indices x_0, y_1, y_2: (a) general form, (b) reduced form.

B. Addition

Let $\mathcal{F}$ and $\mathcal{G}$ be two reduced TDDs over index set I. The summation of $\mathcal{F}$ and $\mathcal{G}$, denoted $\mathcal{F} + \mathcal{G}$, is a reduced TDD with the corresponding tensor $\Phi(\mathcal{F}) + \Phi(\mathcal{G})$. For any $x \in I$ with $x \preceq index(r_\mathcal{F})$ and $x \preceq index(r_\mathcal{G})$, by the TDD version of the Boole-Shannon expansion (cf. Eq. 11), we have

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

Recall here $\mathcal{F}_{x=c}$ (resp. $\mathcal{G}_{x=c}$) is the sub-TDD as defined in Definition 6 for $c \in \{0, 1\}$.

Motivated by this observation, Algorithm 2 implements the *Add* operation for TDDs, in a node-wise manner. The time complexity is $\mathcal{O}(|\mathcal{F}| \cdot |\mathcal{G}|)$, where $|\mathcal{F}|$ and $|\mathcal{G}|$ denote the numbers of nodes in the two TDDs respectively.

Algorithm 2 $Add(\mathcal{F}, \mathcal{G})$

Input: Two reduced TDDs $\mathcal{F}$ and $\mathcal{G}$.
Output: The reduced TDD of $\Phi(\mathcal{F}) + \Phi(\mathcal{G})$.
1: **if** $r_\mathcal{F} = r_\mathcal{G}$ **then**
2:      $tdd \leftarrow \mathcal{F}$
3:      $tdd.weight \leftarrow w_\mathcal{F} + w_\mathcal{G}$
4:      **return** tdd
5: **end if**
6: $x \leftarrow$ the smaller index of $r_\mathcal{F}$ and $r_\mathcal{G}$
7: $tdd \leftarrow$ an empty TDD
8: $tdd.root \leftarrow$ a new node v with index x
9: $v.low \leftarrow Add(\mathcal{F}_{x=0}, \mathcal{G}_{x=0})$
10: $v.high \leftarrow Add(\mathcal{F}_{x=1}, \mathcal{G}_{x=1})$
11: $tdd.weight \leftarrow 1$
12: **return** $reduce(tdd)$

C. 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 $\mathcal{F}$ and $\mathcal{G}$ be two reduced TDDs over I, and var a subset of I denoting the variables to be contracted. Write cont for both tensor and TDD contractions. For any $x \in I$ with

$x \preceq index(r_\mathcal{F})$ and $x \preceq index(r_\mathcal{G})$, we have by definition Eq. 2 that if $x \in var$, then $\mathsf{cont}(\Phi(\mathcal{F}), \Phi(\mathcal{G}), var)$ equals

$$\sum_{c=0}^{1} \mathsf{cont}(\Phi(\mathcal{F}_{x=c}), \Phi(\mathcal{G}_{x=c}), var \backslash \{x\});$$

otherwise, it equals

$$\overline{x} \cdot \mathsf{cont}(\Phi(\mathcal{F}_{x=0}), \Phi(\mathcal{G}_{x=0}), var) \\ + x \cdot \mathsf{cont}(\Phi(\mathcal{F}_{x=1}), \Phi(\mathcal{G}_{x=1}), var).$$

Algorithm 3 gives the detailed procedure for TDD contraction. The time complexity is $\mathcal{O}(|\mathcal{F}|^2 \cdot |\mathcal{G}|^2)$, while $|\mathcal{F}|$ and $|\mathcal{G}|$ denote the numbers of nodes in $\mathcal{F}$ and $\mathcal{G}$, respectively.

To conclude this section, we would like to point out that the *tensor product* of two TDDs $\mathcal{F}$ and $\mathcal{G}$ with disjoint essential indices can be regarded as a special case of contraction. In particular, we have

$$\Phi(\mathcal{F} \otimes \mathcal{G}) = \mathsf{cont}(\Phi(\mathcal{F}), \Phi(\mathcal{G}), \emptyset),$$

and the time complexity of using Algorithm 3 to compute $\mathcal{F} \otimes \mathcal{G}$ becomes $|\mathcal{F}| \cdot |\mathcal{G}|$.

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

Algorithm 3 $\mathsf{cont}(\mathcal{F}, \mathcal{G}, var)$

Input: Two reduced TDDs $\mathcal{F}$ and $\mathcal{G}$, and the set var of variables to be contracted.
Output: The reduced TDD obtained by contracting $\mathcal{F}$ and $\mathcal{G}$ over var.
1: **if** both $\mathcal{F}$ and $\mathcal{G}$ are trivial **then**
2:      $tdd \leftarrow \mathcal{F}$
3:      $tdd.weight \leftarrow w_\mathcal{F} \cdot w_\mathcal{G} \cdot 2^{len(var)}$
4:      **return** tdd
5: **end if**
6: $x \leftarrow$ the smaller index of $r_\mathcal{F}$ and $r_\mathcal{G}$
7: $L \leftarrow \mathsf{cont}(\mathcal{F}_{x=0}, \mathcal{G}_{x=0}, var \backslash \{x\})$
8: $R \leftarrow \mathsf{cont}(\mathcal{F}_{x=1}, \mathcal{G}_{x=1}, var \backslash \{x\})$
9: **if** $x \in var$ **then**
10:      **return** $Add(L, R)$
11: **else**
12:      $tdd \leftarrow$ an empty TDD
13:      $tdd.root \leftarrow$ a new node v with index x
14:      $v.low \leftarrow L$
15:      $v.high \leftarrow R$
16:      $tdd.weight \leftarrow 1$
17:      **return** $reduce(tdd)$
18: **end if**

VI. TWO PARTITION SCHEMES

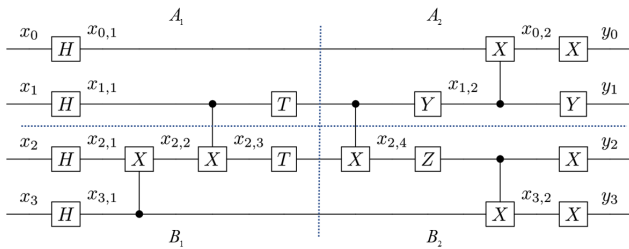Unlike QMDD and QuIDD, the TDD of an input quantum circuit can be calculated very flexibly. In particular, there is

Fig. 11. Partition Scheme I, where only one CX cut is allowed each time.

Fig. 12. Partition Scheme II, where only one CX cut is allowed each time and part C involves up to 2 qubits.

no need to expand a quantum gate to an n-qubit form (by tensoring an identity matrix) when processing it during the simulation of the circuit. In general, the TDD representation of a quantum circuit can be obtained by contracting the TDDs of the individual gates in the circuit in any order. For quantum circuits with low tree-width, following an optimal contraction order will enable us to obtain the TDD representation in polynomial time [18].

In this paper, we use the original qubit order (or its inverse), scan the circuit qubit by qubit, and then rank the indices following the circuit 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 $x \prec x'$ 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. 11 is

$$\prec := (x_0, x_{0,1}, x_{0,2}, y_0, x_1, x_{1,1}, x_{1,2}, y_1, x_2, ..., y_2, x_3, ..., y_3).$$

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 qubit wires are divided into two parts, the upper and the bottom) and then cuts it *vertically* whenever k CNOT gates are separated by the horizontal cut, where k is chosen to ensure that the rank of each part of the circuit is smaller than $2n$.

Example 8. *Consider the circuit shown in Fig. 11 and set $k = 1$, i.e., we allow only one CX cut at a time. The circuit is divided into four parts as shown by the dotted lines. In the contraction process, we first calculate the TDDs of the four parts separately. Then, contracting the left (right, resp.) two TDDs gives the TDD of the left (right, resp.) half of the circuit. Finally, we contract these two TDDs and obtain the TDD of the whole circuit. If we set $k = 2$, then no vertical cut is required and the circuit is partitioned into two parts: the top half and the bottom half. The same TDD can be obtained by contracting the TDDs of the top and the bottom halves.*
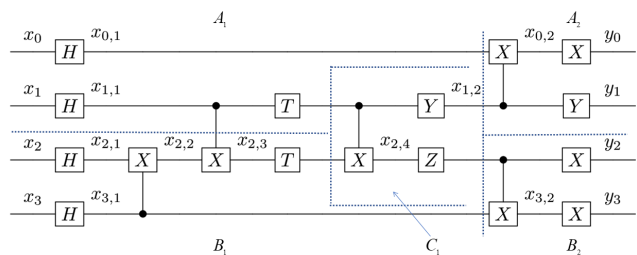
When there are too many CX cuts, the above partition scheme is not efficient as we may need to contract too many high rank tensors. Our second partition scheme allows us to circumvent this obstacle by introducing fewer vertical partitions. First, we horizontally divide the circuit from the middle. However, every time when we have accumulated k_1 CX cuts, we allow a third small block, i.e., an extra C part, involving at most k_2 qubits in the middle of the circuit. When the number of qubits of the C part reaches k_2, we introduce a vertical cut as in the first scheme. The second scheme is a generalisation of the scheme used in [17], where no vertical partition is introduced.

Example 9. *Consider the circuit given in Fig. 11 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. 12. We then compute and contract the TDDs in the order of A, B, 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. 11 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 ≤ 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 ≤ 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 ≤ 4.

VII. NUMERICAL RESULTS

To demonstrate the effectiveness of TDD as an efficient data structure for the representation and manipulation of quantum functionalities, we experimentally simulated a set of 100 benchmark circuits using TDD with the two partition schemes introduced in Sec. VI. The benchmarks we used are obtained from [33]. The numbers of qubits and gates in these benchmarks range from 3 to 16 and 5 to 986, respectively. In our experiments, for the first partition scheme, we set the parameter k as $\lfloor n/2 \rfloor$, where n is the number of qubits in the input circuit. Similarly, for the second partition scheme, we

TABLE I
DATA SUMMARY FOR ALL BENCHMARK CIRCUITS

	QMDD	TDD		
		No part.	Part. Sch. I	Part. Sch. II
Time	5512.16	6201.54	1628.53	1335.99
node num. (final*)	9309	19959	19959	19959
node num. (max*)	24033	49399	26676	25681
ratio (max/final*)	2.58	2.50	1.34	1.29

*Results about the circuit 'cnt3-5_18' are excluded here.

set the two parameters k_1 and k_2 as $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1$, respectively.

We implemented TDD using Python and all experiments were executed on a laptop with Intel i7-1065G7 CPU and 8 GB RAM. Our implementation is far from being optimal. In particular, it assumes a naive and fixed linear order for all TDDs. Nevertheless, in order to provide a rough comparison with the state-of-the-art data structure — QMDD [24], we also implemented QMDD using Python in a straightforward way, without adopting the optimisation techniques used in its C++ implementation [24]. In our experiments, the QMDD of each input circuit is obtained by multiplying them in the circuit order. Our experimental results show that both implementations of QMDD generate the same representation for each benchmark circuit. Source code and detailed empirical results are available at Github.[2]

Table I summarises the experimental results of our TDD implementation on these benchmark circuits and, due to space limitation, we leave the detailed results for individual circuits in the Appendix. We note that, for only one circuit, viz., the 16-qubit circuit 'cnt3-5_180', the QMDD representation is not obtained within 3600 seconds. The same happens when we try to compute its TDD without using partition. Surprisingly, the TDD representation can be obtained within 56 seconds if using either partition scheme. In total, the QMDD calculation takes 5512.16 seconds, which is about 11% less than the time for calculating TDD without using any partition. Meanwhile, the time consumptions for TDD calculation with partition schemes 1 and 2 are just 1628.53 and 1335.99 seconds, which are, respectively, 70% and 75% less than that of QMDD, and 74% and 78% less than that of TDD without partition.

In principle, the TDD representation of a quantum circuit has about twice the number of nodes as the circuit's QMDD. This is because every non-terminal node in a TDD has two successors while a non-terminal node in a QMDD has four. Our experimental results show that the total node number of all final TDDs is 19959, which is just slightly above twice of that of QMDD. As a consequence, this shows that the TDD representation is indeed compact. Furthermore, the simulation time is closely related to the size of QMDDs/TDDs generated in the simulation process. Table I also suggests that TDD simulation with either partition scheme has in average smaller intermediate diagrams than QMDD simulation and TDD simulation without partitions.

[2]https://github.com/XinHong-uts/A-Tensor-Network-Based-Decision-Diagram

VIII. CONCLUSION

We proposed a novel data structure TDD which provides a compact and canonical representation for quantum functionalities and can be used in many design automation tasks, e.g., simulation and equivalence checking, for quantum circuits. This data structure inherits several important properties of tensor networks and supports basic tensor operations like addition and contraction efficiently. It also allows us to compute the representation of quantum circuits in a more flexible way. As demonstrated by empirical experiments on real quantum circuits, by combining with circuit partition, we can reduce the time consumption greatly for some benchmarks when compared with a straightforward python implementation of QMDD.

Future work will consider the following problems. First, our implementation of TDD is far from being optimal. How to optimise the circuit partition scheme or, more general, the contraction order is an interesting open problem. Second, the TDD representations of the same tensor may have quite different sizes if different index orders are used. Take the circuit 'sym6_316.qasm' as an example. The TDD with the original qubit order has only 1228 nodes, while the one reported in Table II, using the inverse order, has 3670 nodes. We will also consider methods of constructing more compact TDDs by searching for good index orders. To this end, a standard method is to gradually transform one TDD into another by changing the order of two adjacent indices. We will improve the performance of the TDD data structure in the follow-up research so that it can be more conveniently applied to the calculation of quantum circuits and tensor networks. A third line of research is to apply the TDD data structure in applications like equivalence checking, simulation, error detection of quantum circuits.

REFERENCES

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, Robert Wille, and Giovanni De Micheli. Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 175–180. IEEE, 2016.

[2] Henrik Reif Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, page 5, 1997.

[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[4] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[5] Jacob Biamonte. Lectures on quantum tensor networks. *arXiv preprint arXiv:1912.10049*, 2019.

[6] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, and Hartmut Neven. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384*, 2017.

[7] Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE design automation conference*, pages 40–45. IEEE, 1990.

[8] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[9] Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *arXiv preprint arXiv:2004.08420*, 2020.

[10] Lukas Burgholzer and Robert Wille. Improved dd-based equivalence checking of quantum circuits. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132. IEEE, 2020.

[11] Jianxin Chen, Fang Zhang, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018.

[12] Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo, and Guo-Ping Guo. 64-qubit quantum circuit simulation. *Science Bulletin*, 63(15):964–971, 2018.

[13] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*, pages 93–108. Springer, 1996.

[14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[15] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *arXiv preprint arXiv:2002.01935*, 2020.

[16] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, et al. Classical simulation of quantum supremacy circuits. *arXiv preprint arXiv:2005.06787*, 2020.

[17] Riling Li, Bujiao Wu, Mingsheng Ying, Xiaoming Sun, and Guangwen Yang. Quantum supremacy circuit simulation on sunway taihulight. *IEEE Transactions on Parallel and Distributed Systems*, 31(4):805–816, 2019.

[18] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[19] Dmitri Maslov, Jin-Sung Kim, Sergey Bravyi, Theodore J Yoder, and Sarah Sheldon. Quantum advantage for computations with limited space. *arXiv preprint arXiv:2008.06478*, 2020.

[20] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, Feb 2016.

[21] D Michael Miller and Mitchell A Thornton. Qmdd: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, pages 30–30. IEEE, 2006.

[22] Paul Molitor and Janett Mohnke. *Equivalence checking of digital circuits: fundamentals, principles, methods.* Springer Science & Business Media, 2007.

[23] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[24] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. Qmdds: Efficient quantum function representation and manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):86–99, 2015.

[25] Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W Draeger, Eric T Holland, and Robert Wisnieff. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867*, 2017.

[26] George F Viamontes, Igor L Markov, and John P Hayes. Improving gate-level simulation of quantum circuits. *Quantum Information Processing*, 2(5):347–380, 2003.

[27] Shiou-An Wang, Chin-Yung Lu, I-Ming Tsai, and Sy-Yen Kuo. An xqdd-based verification method for quantum circuits. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 91(2):584–594, 2008.

[28] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. Efficient and correct compilation of quantum circuits. In *IEEE International Symposium on Circuits and Systems*, 2020.

[29] Shigeru Yamashita and Igor L Markov. Fast equivalence-checking for quantum circuits. In *2010 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 23–28. IEEE, 2010.

[30] Mingsheng Ying. *Foundations of Quantum Programming.* Morgan Kaufmann, 2016.

[31] Mingsheng Ying and Yuan Feng. An algebraic language for distributed quantum computing. *IEEE Transactions on Computers*, 58(6):728–743, 2009.

[32] Alwin Zulehner, Stefan Hillmich, and Robert Wille. How to efficiently handle complex values?: Implementing decision diagrams for quantum computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019.

[33] Alwin Zulehner, Alexandru Paler, and Robert Wille. 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):1226–1236, 2018.

[34] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):848–859, 2018.

APPENDIX
DETAILED EMPIRICAL RESULTS

TABLE II
EXPERIMENT DATA

Benchmarks			QMDD			TDD No. Part.			TDD Part. Sch. I		TDD Part. Sch. II	
Name	Qubit num	Gate num	Time	node num. max	node num. final	Time	node num. max	node num. final	Time	node num. max	Time	node num. max
graycode6_47	6	5	0.02	12	12	0.03	30	30	0.02	30	0.03	30
ex1_226	6	7	0.03	12	12	0.02	22	22	0.02	22	0.02	22
xor5_254	6	7	0.04	12	12	0.02	22	22	0.02	22	0.02	22
4gt11_84	5	18	0.07	22	22	0.07	38	38	0.06	38	0.04	38
ex-1_166	3	19	0.05	10	9	0.06	22	17	0.06	22	0.05	22
4mod5-v0_20	5	20	0.16	28	22	0.12	54	38	0.15	48	0.12	48
ham3_102	3	20	0.05	10	10	0.05	20	20	0.06	20	0.04	20
4mod5-v1_22	5	21	0.17	28	16	0.14	54	34	0.19	54	0.14	54
mod5d1_63	5	22	0.27	34	19	0.23	46	29	0.15	46	0.12	46
4gt11_83	5	23	0.12	26	26	0.14	60	60	0.13	60	0.10	60
4gt11_82	5	27	0.13	26	18	0.16	56	46	0.16	56	0.10	56
rd32-v0_66	4	34	0.21	14	9	0.24	30	21	0.19	28	0.13	30
4mod5-v0_19	5	35	0.31	30	19	0.26	58	39	0.32	50	0.19	50
mod5mils_65	5	35	0.31	26	19	0.25	40	39	0.27	39	0.17	39
3_17_13	3	36	0.12	13	10	0.13	24	20	0.17	24	0.13	24
4mod5-v1_24	5	36	0.25	30	28	0.25	58	50	0.25	58	0.17	58
alu-v0_27	5	36	0.23	42	30	0.35	76	58	0.28	76	0.27	70
rd32-v1_68	4	36	0.19	14	9	0.19	30	21	0.20	28	0.14	30
alu-v1_28	5	37	0.26	38	26	0.27	80	52	0.24	68	0.18	68
alu-v1_29	5	37	0.24	34	24	0.21	70	44	0.22	59	0.18	59
alu-v2_33	5	37	0.37	42	26	0.27	78	50	0.25	65	0.25	76
alu-v3_35	5	37	0.30	40	32	0.34	70	66	0.34	70	0.35	66
alu-v4_37	5	37	0.24	42	34	0.28	76	72	0.31	76	0.25	72
miller_11	3	50	0.18	14	11	0.20	24	21	0.27	24	0.23	24
decod24-v0_38	4	51	0.25	18	16	0.33	36	36	0.42	36	0.40	36
alu-v3_34	5	52	0.32	36	32	0.32	68	66	0.45	66	0.40	66
decod24-v2_43	4	52	0.34	21	16	0.36	36	36	0.46	36	0.28	36
mod5d2_64	5	53	0.25	30	16	0.30	56	36	0.32	56	0.25	56
4gt13_92	5	66	0.57	26	26	0.76	58	58	0.56	58	0.42	58
4gt13-v1_93	5	68	0.84	38	26	1.32	72	58	0.88	72	0.73	72
4mod5-v0_18	5	69	0.53	36	20	0.62	70	42	0.54	60	0.40	70
4mod5-v1_23	5	69	0.68	28	19	0.53	62	41	0.55	62	0.37	62
one-two-three-v2_100	5	69	0.96	82	46	1.13	142	90	0.90	142	0.70	142
4mod5-bdd_287	7	70	1.81	111	74	1.42	175	128	1.33	158	0.98	158
one-two-three-v3_101	5	70	0.71	66	46	0.88	128	90	1.09	127	0.83	127
decod24-bdd_294	6	73	0.42	30	17	0.61	74	43	0.59	62	0.42	62
4gt5_75	5	83	0.91	54	42	1.17	102	86	1.34	102	1.15	102
alu-bdd_288	7	84	1.27	44	34	1.37	94	74	1.28	76	1.08	92
alu-v0_26	5	84	0.63	36	23	1.00	68	48	1.10	64	1.35	64
rd32_270	5	84	0.79	24	13	0.83	52	31	0.80	48	0.66	52
decod24-v1_41	5	85	0.56	26	14	0.65	50	29	0.96	50	0.71	50
4gt5_76	5	91	0.69	42	20	1.03	92	48	1.11	92	0.88	92
4gt13_91	5	103	0.63	42	28	1.11	84	64	1.20	84	1.11	84
4gt13_90	5	107	0.68	42	20	1.17	84	48	1.11	84	1.07	84
alu-v4_36	5	115	0.71	36	32	1.26	68	68	1.05	68	1.01	68
4gt5_77	5	131	1.34	44	27	1.81	78	61	1.42	78	1.55	69
one-two-three-v1_99	5	132	0.88	43	43	1.74	85	85	1.29	85	1.38	85
rd53_138	8	132	8.28	138	76	6.60	275	167	4.09	219	3.45	167
one-two-three-v0_98	5	146	1.76	58	42	2.26	110	82	2.08	110	1.84	106
4gt10-v1_81	5	148	1.38	40	33	1.96	80	68	1.94	74	2.07	74
decod24-v3_45	5	150	1.34	35	17	1.89	74	36	1.90	73	2.17	74
aj-e11_165	5	151	2.66	54	19	2.48	104	40	2.52	104	2.46	104
4mod7-v0_94	5	162	1.32	47	29	2.14	91	62	1.74	91	1.62	91
alu-v2_32	5	163	1.20	57	22	1.77	103	52	1.56	97	1.59	103
4mod7-v1_96	5	164	1.49	46	29	2.26	88	62	1.95	80	1.97	79
mini_alu_305	10	173	27.67	866	375	26.30	1730	793	15.42	966	13.32	1730
cnt3-5_179	16	175	273.78	53	48	170.63	109	100	10.88	143	9.97	143
mod10_176	5	178	1.92	50	15	2.45	96	31	2.37	96	2.17	96
4gt4-v0_80	6	179	2.61	74	33	3.59	136	69	2.60	136	2.66	136
4gt12-v0_88	6	194	2.51	102	37	5.02	182	75	3.60	152	3.22	154
qft_10	10	200	32.66	20	11	45.83	55	31	10.20	49	9.26	49
0410184_169	14	211	50.13	57	39	26.45	130	81	13.14	97	9.20	113
sys6-v0_111	10	215	40.22	473	247	38.73	898	535	24.02	681	15.16	536

4_49_16	5	217	3.00	50	22	3.91	102	45	3.77	77	3.18	77
4gt12-v1_89	6	228	2.03	50	25	3.89	88	57	4.26	92	3.48	114
rd73_140	10	230	33.35	242	157	35.87	565	379	21.74	379	17.53	443
4gt4-v0_79	6	231	3.41	102	31	5.10	176	69	5.92	116	3.75	156
hwb4_49	5	233	3.20	58	23	4.16	110	47	4.62	98	4.35	98
4gt4-v0_78	6	235	3.21	90	23	5.08	172	55	5.53	116	3.70	156
mod10_171	5	244	3.20	44	17	4.11	86	36	4.22	86	3.59	86
4gt12-v0_87	6	247	3.61	60	31	5.72	142	71	6.02	132	3.66	119
4gt12-v0_86	6	251	3.49	60	23	5.72	142	55	6.00	132	4.09	119
4gt4-v0_72	6	258	3.19	68	31	6.38	133	67	5.40	128	4.84	128
sym6_316	14	270	132.91	5608	1520	199.55	13460	3670	67.41	3670	67.70	3670
4gt4-v1_74	6	273	3.97	82	30	8.37	154	67	6.98	154	6.77	154
rd53_311	13	275	123.18	1332	684	157.22	2836	1536	97.13	2652	57.59	1536
mini-alu_167	5	288	2.95	58	19	5.29	116	41	5.16	116	4.61	116
one-two-three-v0_97	5	290	2.72	55	41	5.89	103	80	5.00	96	4.36	96
rd53_135	7	296	7.99	145	36	17.67	313	78	11.45	221	11.36	221
ham7_104	7	320	3.47	130	130	6.00	238	238	5.24	238	4.65	238
sym9_146	12	328	99.46	719	229	131.88	1738	595	75.86	595	72.93	595
mod8-10_178	6	342	5.03	70	18	8.91	140	40	8.33	117	7.65	136
rd84_142	15	343	722.52	9753	3588	827.38	18111	7115	393.61	8431	405.36	8045
4gt4-v0_73	6	395	6.31	66	23	13.33	148	55	9.85	144	9.33	130
ex3_229	6	403	5.36	44	19	9.67	94	44	10.77	114	8.66	94
mod8-10_177	6	440	7.03	69	21	14.06	147	47	13.20	147	11.79	147
alu-v2_31	5	451	3.51	51	18	9.98	88	44	9.12	88	9.04	88
C17_204	7	467	11.36	82	25	21.44	160	57	19.02	160	19.24	160
rd53_131	7	469	18.07	205	34	38.73	435	82	26.77	387	29.14	322
cnt3-5_180	16	485	>3600.00	-	-	>3600.00	-	-	55.75	296	38.05	286
alu-v2_30	6	504	7.61	103	30	26.57	199	62	24.32	195	23.25	195
mod5adder_127	6	555	9.42	97	51	27.70	190	107	21.01	190	17.48	182
rd53_133	7	580	13.56	73	26	33.79	144	58	20.48	124	19.31	124
majority_239	7	612	7.76	112	16	32.19	218	39	21.97	218	21.79	173
ex2_227	7	631	11.49	106	22	34.64	208	50	30.66	204	28.96	181
cm82a_208	8	650	9.85	114	31	49.94	252	77	68.04	215	27.99	171
sf_276	6	778	11.50	48	19	38.92	107	42	51.83	98	28.78	107
sf_274	6	781	12.24	50	20	41.01	110	44	54.80	102	29.55	110
con1_216	9	954	25.81	178	37	172.95	408	91	179.09	394	118.79	244
wim_266	11	986	113.39	135	62	222.21	333	160	143.13	333	95.99	324
SUM			5512.16	24033	9309	6201.54	49399	19959	1628.53	26776	1335.99	25681

*In our experiments, if the executing time exceeds 3600 seconds, we stopped it.

*In the summation, the terms greater than 3600 were calculated as 3600.

*The node numbers of the circuit 'cnt3-5_18' was not counted in the SUM for all the columns.