entropy

Article

Contraction Heuristics for Tensor Decision Diagrams

Christian Bøgh Larsen [†], Simon Brun Olsen [†], Kim Guldstrand Larsen and Christian Schilling [*]

Department of Computer Science, Aalborg University, 9220 Aalborg, Denmark; kgl@cs.aau.dk (K.G.L.)
[*] Correspondence: christianms@cs.aau.dk
[†] These authors contributed equally to this work.

Abstract: In this paper, we study the equivalence problem for quantum circuits: Given two quantum circuits, are they equivalent? We reduce this problem to the contraction problem of a tensor network. The order in which the contraction operations between tensors are applied has a crucial impact on efficiency, which is why many heuristics have been proposed. In this work, we use an efficient representation of tensors as a tensor decision diagram. Since existing contraction heuristics do not perform well in combination with these diagrams, we propose two new contraction heuristics. We demonstrate experimentally that our heuristics outperform other state-of-the-art heuristics. We also demonstrate that our framework yields state-of-the-art performance for equivalence checking.

Keywords: tensor network; contraction planning; tensor decision diagram; heuristics; quantum circuit; equivalence checking

1. Introduction

Interest in quantum computing has rapidly increased in both the scientific community and industry. Quantum computers promise to speed up certain computations to a potentially exponential degree [1]. While quantum computers are currently impractical for industrial use, the rate of improvement is similar to that of early classical computers, and it is expected that in the near future, practically usable quantum computers will exist. In order to enable effective use of quantum computers once they mature, the required tools for supporting quantum computation need to be developed now, e.g., verification of quantum algorithms against specifications [2,3] or efficient simulation [4,5].

A central task in quantum computing is the design and optimization of quantum algorithms, with the prevailing paradigm being quantum circuits. The circuit design typically takes place at a high abstraction level, where arbitrary quantum gates can be used. Since a real quantum computer only has a limited set of quantum gates available, the circuit must then be transformed, or *compiled*, into an equivalent circuit meeting these constraints [6]. In addition, the size of a circuit correlates with the noise that occurs during computation. Since noise is the main challenge to the practical use of quantum computers today, additional optimization steps are applied to reduce the size of the circuit [7]. Quantum compilers thus perform an important and nontrivial task, and it is critical that the circuit they produce is functionally equivalent to the original design [8].

Functional equivalence of two quantum circuits means that, for any input state, the two circuits agree on the resulting output state. There are strong reasons to believe that checking for equivalence is hard: deciding approximate equivalence is QMA-complete [9,10], and deciding exact equivalence is NQP-complete [11]; problems in these complexity classes are widely believed to require exponential computations in the worst case.

In this work, we study the problem of checking the equivalence of quantum circuits. Our work is inspired by recent developments in the field, which we briefly summarize next.

Citation: Larsen, C.B.; Olsen, S.B.; Larsen, K.G.; Schilling, C. Contraction Heuristics for Tensor Decision Diagrams. *Entropy* **2024**, *26*, 1058. https://doi.org/10.3390/e26121058

Academic Editors: Giuliano Benenti, Xiao Yuan, Xiaoming Zhang and Bálint Koczor

Received: 4 November 2024
Revised: 24 November 2024
Accepted: 3 December 2024
Published: 5 December 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1.1. Related Work

Tensor networks. A *tensor* is a higher-dimensional generalization of a matrix. A tensor network is a graph with tensors as nodes. As such, a quantum circuit can be naturally modeled as a tensor network where every node is a matrix representing a quantum gate [12].

The most important operation in tensor networks is the contraction of two tensors into one. Iteratively contracting tensors in the network eventually yields a single tensor. Since the order in which the contractions are applied has a large impact on the intermediate size, many heuristics have been proposed to find a good contraction order [13,14], several of which are implemented in the contraction tool COTENGRA [15].

Decision diagrams. A quantum gate matrix can be represented more succinctly in a symbolic data structure called a *decision diagram*. Several decision diagrams have been proposed in the literature [16–20], and we refer to them uniformly as quantum decision diagrams (QDDs). QDDs may offer exponential compression, but they are no silver bullet. Since QDDs exploit symmetries in the matrix, different matrices/circuits show different compression potential. Analogously, a *tensor decision diagram* (TDD) symbolically represents a tensor [21], which thus generalizes the QDD. TDDs have recently been demonstrated to speed up quantum simulations [4].

Quantum circuit equivalence. Checking the equivalence of two quantum circuits C_1 and C_2 is a conceptually simple comparison of their characteristic matrices. However, the size of these matrices is exponential in terms of the number of qubits, and thus, this comparison cannot be performed for reasonably sized circuits. Alternatively, the problem reduces to comparing the composition of C_1 and the adjoint of C_2 to the identity [9]. By representing gates as QDDs, this "adjoint scheme" can lead to an efficient equivalence check [22]. By performing the involved matrix multiplications in a clever order, this scheme performs even better [23], and it is implemented in the tool QCEC [24]. Efforts to identify a good order using contraction heuristics from tensor networks were not successful [25], at least when restricted to QDDs; in this work, we instead use TDDs for representing tensor networks efficiently, and we shall see that this yields a method that is competitive with the QDD-based approach. Equivalence of *dynamic* quantum circuits was addressed by directly encoding the two circuits as TDDs individually and then checking whether these TDDs are identical, i.e., not using the adjoint scheme that we use [26].

Orthogonal approaches to equivalence checking include ZX-calculus [27], which can sometimes yield results quickly [28]. Alternatively, if the quantum circuits belong to a restricted subclass called the Clifford group, a polynomial-time approach exists [29]. TDDs have been applied to approximate equivalence checking based on computing traces [30].

1.2. Contributions

In this paper, we integrate several of the above techniques into a new approach to the equivalence checking of quantum circuits. We represent the quantum circuits as a tensor network with the adjoint scheme described above. When operating on tensor networks, further benefits arise because tensors of different sizes can be contracted, which is not possible with QDDs. To efficiently represent the tensors in the network, we use TDDs.

However, as mentioned above, decision diagrams are not equally efficient on different circuits. Hence, contraction orders that may be beneficial for tensor networks with ordinary tensors may be suboptimal, or even detrimental, to TDD networks (i.e., tensor networks where the tensors are represented as TDDs). Thus, we are in need of new contraction heuristics for TDD networks, and we propose two such heuristics in this work. In our experimental evaluation, we demonstrate that our heuristics outperform other heuristics implemented in COTENGRA and that our TDD-based approach for equivalence checking is competitive with QCEC.

1.3. Outline

The rest of this paper is structured as follows: In Section 2, we summarize background information on quantum computing, tensor networks, decision diagrams, and contraction heuristics. In Section 3, we describe our contraction heuristics. In Section 4, we evaluate our contraction heuristics and compare them to state-of-the-art methods. In Section 5, we conclude the paper and discuss future directions.

2. Background

In this section, we recall some basic concepts needed to understand the rest of the paper. We start with a short introduction to quantum circuits. Then, we explain tensor networks and how they relate to quantum circuits, followed by describing tensor contraction. After that, we give a brief overview of tensor decision diagrams. Finally, we discuss how to check the equivalence of two quantum circuits.

2.1. Quantum Circuits

We briefly recall the main concepts of quantum computing and the quantum circuit model. For a broader introduction, we refer to the literature [31].

The basic unit in quantum computing is the *qubit*, like the bit is to digital computing. While a bit can be in one of two states, 0 or 1, a qubit can be in the basis states $|0\rangle$ or $|1\rangle$ (using the Dirac/BraKet notation), or it can be in a linear combination of these; in the latter case, the qubit is said to be in a superposition. Formally, a qubit $|q_0\rangle$ is described as $|q_0\rangle = \alpha|0\rangle + \beta|1\rangle$. The values $\alpha, \beta \in \mathbb{C}$ are the amplitudes, and they must satisfy the side condition $|\alpha|^2 + |\beta|^2 = 1$. Measuring $|q_0\rangle$ results in the basis state $|0\rangle$ with probability $|\alpha|^2$ and in the basis state $|1\rangle$ with probability $|\beta|^2$.

We generalize the notion of a *quantum state* to multiple qubits. Since a quantum state may be in a superposition of each combination of basis states, it has 2^n amplitudes for n qubits. A quantum state can also be represented as a state vector of the amplitudes (e.g., $[\alpha, \beta]^\mathsf{T}$ for $|q_0\rangle$ above).

Example 1 (Quantum state)**.** *Consider a quantum state of 3 qubits, q_0, q_1, and q_2. A state vector for such a state is*

$$[\alpha_{000}, \alpha_{001}, \ldots, \alpha_{111}]^\mathsf{T} \qquad s.t. \sum_i |\alpha_i|^2 = 1,$$

which represents a linear combination of all eight basis states:

$$|q_0 q_1 q_2\rangle = \alpha_{000}|000\rangle + \alpha_{001}|001\rangle + \cdots + \alpha_{111}|111\rangle$$

The probability of measuring the basis state $|i\rangle$ is $|\alpha_i|^2$.

Quantum computation aims at transforming quantum states. A common framework to express these transformations is the quantum circuit model. At the lowest level, a *quantum gate* (henceforth gate) transforms a quantum state into a new quantum state. It is generally sufficient to restrict oneself to gates involving at most two qubits. Each gate corresponds to the application of a linear map with an associated complex unitary matrix.

Example 2 (Quantum gate)**.** *Given a quantum state over a single qubit q_0, the Hadamard gate H has the associated matrix (we abuse notation and write H both for the gate and the associated matrix).*

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The Hadamard gate transforms the basis state $|0\rangle = 1 \cdot |0\rangle + 0 \cdot |1\rangle$ into a superposition:

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Similar to a classical digital circuit, we can combine quantum gates into a *quantum circuit* (henceforth circuit). Figure 1a shows an example circuit involving three gates on two qubits. In the example, the CNOT gate acts on two qubits, while the H gate acts on only one qubit. To compute the effect of the CNOT gate on a two-qubit vector (and hence a four-dimensional state vector) $|q_0\rangle$, we can simply multiply with the associated 4×4-matrix CX as before: $CX \cdot |q_0\rangle$. However, since the H gate only acts on one qubit, we cannot apply matrix multiplication directly. As an intermediate step, we insert an artificial identity matrix on the lower qubit wire, as shown in Figure 1b.

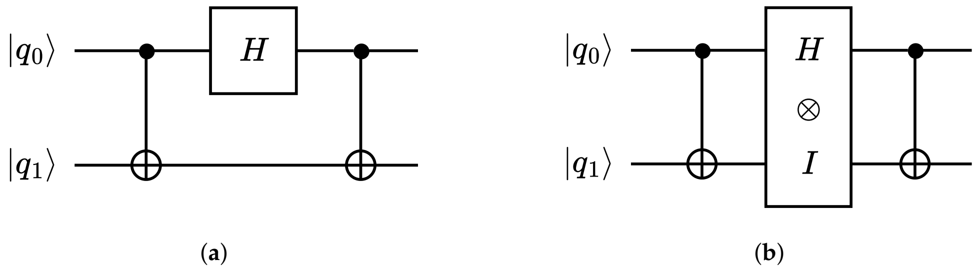


Figure 1. A simple quantum circuit. (**a**) A two-qubit quantum circuit consisting of two CNOT-gates and one Hadamard gate; (**b**) the same circuit after expanding the Hadamard gate H using the tensor product.

In general, we need to compute the effect of multiple parallel gates with qubit counts $n_1, \ldots, n_k$, respectively, to a state vector involving n qubits, where $n_1 + \cdots + n_k = n$. This is achieved by taking the tensor product of the gate matrices.

Example 3 (Quantum circuit)**.** *The effect of applying a Hadamard gate to the first qubit of a two-qubit system (as in Figure 1) can be computed as the tensor product*

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}.$$

Thus, the whole effect of the circuit in Figure 1a is $CX \cdot (H \otimes I) \cdot CX$, or

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix}.$$

2.2. Tensor Networks

Matrices are sufficient to express quantum gates (and circuits). A *tensor* is a generalization to higher dimensions. We can thus use tensors to represent both quantum gates (two-dimensional matrices) and quantum states (one-dimensional vectors).

Tensors have an associated index vector $\mathcal{I}$, where each entry is an *index variable* for the corresponding dimension. In our context, a tensor will have an index for each qubit, and the corresponding index variable has two possible values, one for each of the two basis states ($|0\rangle$ and $|1\rangle$). Figure 2 shows a three-dimensional tensor together with its indices.

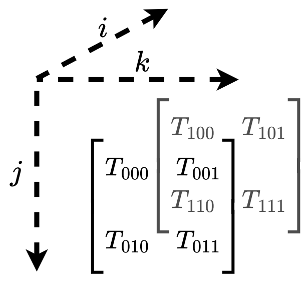
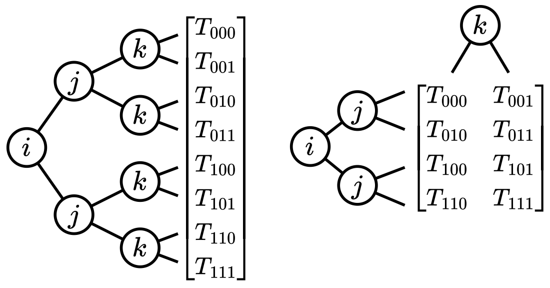


Figure 2. Example of a three-dimensional tensor with corresponding indices i, j, and k.

The elements in our tensors are complex numbers, so tensors have the form $T \in \mathbb{C}^{2^{|\mathcal{I}|}}$. We will use lowercase symbols to denote a concrete index vector, e.g., $\vec{i} \in \{0,1\}^{|\mathcal{I}|}$.

Example 4 (Tensor)**.** *Let $\mathcal{I} = [i, j, k]^\intercal$ be a three-dimensional vector of index variables. Consider the tensor T depicted in Figure 3, which is shown in two alternative (but equivalent) representations. A (concrete) index vector is $\vec{i} = [0, 0, 0]^\intercal$, with the corresponding tensor entry $T_{\vec{i}} = T_{000}$.*



i
j
k
T_{000} T_{001} T_{010} T_{011} T_{100} T_{101} T_{110} T_{111}

Figure 3. A tensor with three index variables *i*, *j*, and *k*, equivalently depicted both as a column vector and as a matrix, showing how indices correspond to the position of the elements.

As shown in Figure 3, the placement of elements is determined by the indices, and the alignment with rows and columns does not have any meaning for the tensor.

Just as gates can be combined into quantum circuits, we can combine multiple tensors into *tensor networks*. A tensor network is a graph whose vertices are tensors and whose edges connect tensors if they share an index. The same index can only be shared by two tensors, and we hence use the words *edges* and *indices* interchangeably in this context.

Circuits contain input and output wires which are only connected to one gate, the other end being free (unless a state vector is supplied as an input). Similarly, our tensor networks also have dedicated input and output indices which are only connected to one tensor; we refer to these as outer edges. Formally:

Definition 1 (Tensor network)**.** *A tensor network is an undirected graph $G = (V, E, E_{outer}, S)$, where $V \subseteq \bigcup_{m \in \mathbb{N}} \mathbb{C}^{2^m}$ is the finite set of vertices, each of which is a tensor of some index dimension m; S is the finite set of index variables; $E \subseteq V \times V \times 2^S$ is a set of (inner) edges; and $E_{outer} \subseteq V \times 2^S$ is a set of outer edges. Each edge is labeled with a nonempty set of index variables.*

Example 5 (Tensor network)**.** *Consider the tensor network (V, E, E_{outer}, S) depicted in Figure 4a. It consists of three tensors: $V = \{CX_{fgjk}, H_{gh}, CX_{hikl}\}$. The tensors are the matrices corresponding to the respective gates, as indicated by the names. (For convenience, we use the naming convention that a tensor is named after its gate with its indices in the subscript.) The index set S of the tensor network is determined by the possible index variables of all tensors in the tensor network, here $S = \{f, g, h, i, j, k, l\}$. The outer edges E_{outer} are the edges that only attach to one tensor, i.e., those labeled with f, i, j, and l. The remaining edges each connect two of the tensors. For our tensor network, we have the following sets of edges:*

$$E = \{(CX_{fgjk}, H_{gh}, \{g\}), (CX_{fgjk}, CX_{hikl}, \{k\}), (H_{gh}, CX_{hikl}, \{h\})\}$$
$$E_{outer} = \{(CX_{fgjk}, \{f\}), (CX_{fgjk}, \{j\}), (CX_{hikl}, \{i\}), (CX_{hikl}, \{l\})\}$$

Similarly to how the gate matrices can be combined into a single matrix representation of a circuit by matrix multiplication, the tensors of the tensor network can be combined into a single tensor representing the same circuit. The operation for combining tensors is *tensor contraction*, which generalizes matrix multiplication. The operation multiplies the elements of each tensor with corresponding index values together, summing over shared indices.

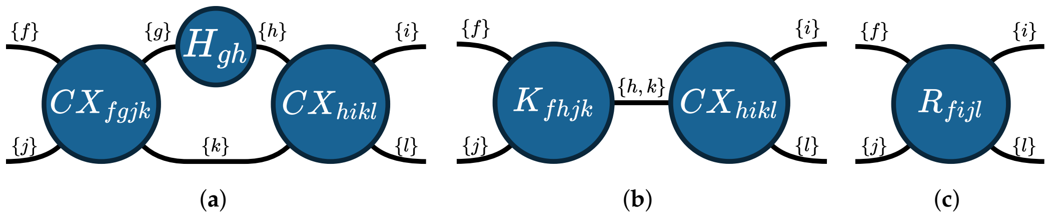


Figure 4. The tensor network of the circuit in Figure 1a being contracted into a single tensor in two contraction steps. The edges between two tensors are labeled with the shared indices. (**a**) Original tensor network; (**b**) contraction of CX_{fgjk} and H_{gh}; (**c**) final result.

Definition 2 (Tensor contraction)**.** *Let $T_{\vec{I},\vec{J}}$ and $T'_{\vec{J},\vec{K}}$ be two tensors with their index variables split into shared variables $\vec{J}$ and unique variables $\vec{I}$ and $\vec{K}$, respectively. The result of the* (tensor) contraction *is the tensor*

$$R_{\vec{i},\vec{j}} = \sum_{\vec{x}\in\{0,1\}^{|\vec{J}|}} T_{\vec{i},\vec{x}} \cdot T'_{\vec{x},\vec{j}}.$$

Example 6 (Tensor contraction)**.** *Recall the tensor network from Figure 4a. By contracting the tensors CX_{fgjk} and H_{gh} over the $\{g\}$ edge, we obtain the tensor (depicted in Figure 4b)*

$$K_{fhjk} = \sum_{x\in\{0,1\}} CX_{fxjk} \cdot H_{xh}.$$

Next, by contracting the tensors K_{fhjk} and CX_{hikl} over the $\{h,k\}$ edge, we obtain the tensor (depicted in Figure 4c)

$$R_{fijl} = \sum_{x,y\in\{0,1\}} K_{fxjy} \cdot CX_{xiyl}.$$

2.3. Tensor Decision Diagrams

A tensor decision diagram (TDD) is a symbolic representation of a tensor. TDDs exploit symmetries to avoid storing (multiples of) the same tensor entries several times.

Definition 3 (Tensor decision diagram [21])**.** *A* tensor decision diagram *is a weighted, rooted, and directed acyclic graph (V, E, idx, w_g) over a set of indices S, defined as follows:*

- *$V = V_N \cup \{v_T\}$ is the set of nodes, where V_N is a finite set of non-terminal nodes, and v_T is the terminal node.*
- *$E \subseteq V_N \times V \times \mathbb{C} \times \{low, high\}$ is the set of weighted edges. Each node in V_N has exactly one outgoing low edge and one outgoing high edge.*
- *$idx : V_N \to S$ assigns an index from the index set S to each non-terminal node.*
- *$w_g \in \mathbb{C}$ is a global weight associated with the root node.*

While there are multiple TDD representations of the same tensor, in practice, one uses a normalization procedure such that for each tensor there exists a corresponding canonical TDD. For further details on TDDs, we refer to the original presentation [21].

Example 7. *Figure 5 shows the TDDs for the tensors from Figure 4 before and after contraction. The representation size after contraction (7 nodes) is smaller than before contraction ($8+3$ nodes).*

2.4. Quantum Circuit Equivalence

We now turn to the equivalence problem for quantum circuits. First, we consider again the quantum circuit model, and later generalize the idea to tensor networks.

Recall that each circuit corresponds to the application of a single unitary matrix. In this light, we define the equivalence of two quantum circuits via their corresponding matrices.

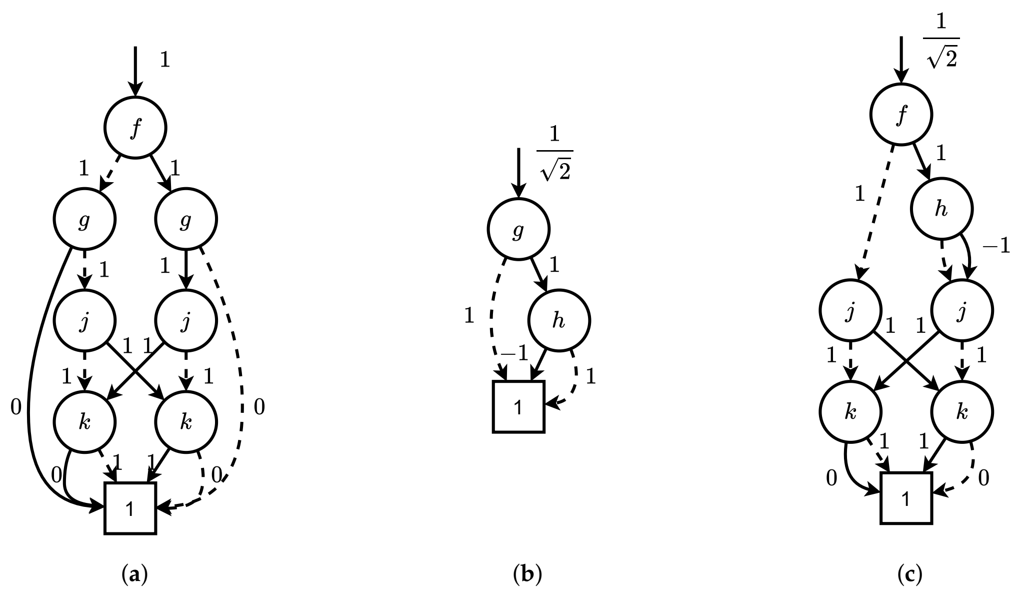


Figure 5. TDDs of two tensors in Figure 4, and the resulting TDD after contraction. (**a**) TDD for CX_{fgjk}; (**b**) TDD for H_{gh}; (**c**) TDD for K_{fhjk}.

Definition 4 (Quantum circuit equivalence)**.** *Two circuits C_1 with n_1 qubits and C_2 with n_2 qubits and with matrix representations U_{C_1} and U_{C_2}, respectively, are* functionally equivalent, *written $C_1 \equiv C_2$, if and only if $n_1 = n_2$, and we have that*

$$\exists\theta \in [0, 2\pi) \colon U_{C_1} = e^{i\theta} \cdot U_{C_2}.$$

Note that the matrices only need to be identical up to a factor $e^{i\theta}$ called the global phase; this is due to the quantum phenomenon that two states can only be experimentally distinguished up to this factor.

However, computing the matrices of two n-qubit circuits is expensive, since the matrices have $2^n \times 2^n$ (i.e., exponentially many) entries. We thus turn to potentially cheaper ways of determining equivalence.

Since unitary matrices are invertible, all gates (and hence circuits) are reversible. We denote the adjoint of a matrix U by $U^\dagger$. Then, we get the following alternative [22]:

Proposition 1. *Given two n-qubit circuits C_1 and C_2, we have:*

$$C_1 \equiv C_2 \iff \exists\theta \in [0, 2\pi) \colon U_{C_1} \cdot U^\dagger_{C_2} = e^{i\theta} \cdot I.$$

In other words, we can multiply the first circuit with the adjoint of the second circuit and compare the result to the identity matrix. (To simplify the presentation, we ignore the global phase in the following and only compare to the identity.)

While there is no immediate advantage in using this combined circuit setup as opposed to checking whether the matrix representations of two circuit are equivalent, there is a possibility of exploiting the fact that gates from one circuit are negated by gates from the inverse of the other circuit. In particular, if C_1 and C_2 are identical, then the last gate of C_1 and the first gate of C_2 cancel out, and this holds true for all other pairs of gates.

Additionally, there exist alternative data structures (quantum decision diagrams (QDDs)) to represent quantum matrices, potentially avoiding the exponential space requirements. Hence, by performing matrix multiplication on gates from both circuits, we may be able to maintain such a data structure without an exponential blow-up and thus obtain an efficient procedure for checking the equivalence of quantum circuits. It should be noted that there is no guarantee that this blow-up can generally be avoided. However, it has been demonstrated that it can be avoided in many practical cases [23].

For tensors and tensor networks, we can apply a similar approach. Initially, we transform the two circuits C_1 and C_2 into two tensor networks. This is a standard construction (e.g., [32]) using the matrix as the tensor for each gate. Next, the straightforward approach would be to contract both tensor networks and finally compare the two resulting tensors for equality (up to a global phase). Again, we can apply the same trick as in Proposition 1 and build a single tensor network with C_2 inverted. Then, we contract that tensor network and in the end check whether the resulting tensor is the identity (up to a global phase).

As with quantum circuits, we do not necessarily gain anything from this construction if we use an explicit tensor representation. However, just as with QDDs, when representing tensors with TDDs, we can often avoid an exponential blow-up in practice.

Note that working with tensors as compared to quantum gates has potential benefits. Gates are represented as matrices, and thus, contraction corresponds to matrix multiplication, for which the two matrices need to have the same dimension. General tensor contraction can be performed on any subset of shared indices. This allows one to partially contract tensors resulting from parallel gates in a quantum circuit, whereas a gate-based approach has to multiply with all of the gates at the same time. For example, in Figure 4, we contracted a CX tensor with an H tensor, whereas a gate-based approach would have to insert an identity gate (using a tensor product) as in Figure 1b.

2.5. *Contraction Heuristics for Tensor Networks and TDD Networks*

Recall that, to contract a tensor network to a single tensor, we just have to repeatedly apply the contraction operation for two tensors with shared indices. The sequence in which tensors are contracted is called the *contraction plan*. Each contraction step will reduce the number of tensors in the network by one, thus eventually resulting in one tensor representing the whole network. This final tensor is unique, independent of the contraction plan, but the intermediate tensors are not. Thus, the contraction plan has a large impact on the size of the intermediate representation. Correspondingly, a significant amount of work has gone into finding good heuristics to choose a good contraction plan.

Example 8 (Tensor contraction (alternative))**.** *Recall the tensor network from Figure 4a. In Example 6, we chose one of three possible contraction plans. The two alternatives are*

1. *Contract CX_{hikl} and H_{gh} via edge h to obtain K_{gikl}. Then contract CX_{fgjk} and K_{gikl} via edges $\{g, k\}$.*
2. *Contract CX_{fgjk} and CX_{hikl} via edge k to obtain K_{fgjhil}. Then contract H_{gh} and K_{fghjil} via edges $\{g, h\}$.*

Existing heuristics have been designed under the assumption that the tensors are explicitly stored as high-dimensional vectors. In this work, we instead represent tensors implicitly as TDDs. TDDs representing tensors with the same number of elements may have vastly different representation sizes. Hence, it is not clear whether the existing heuristics will perform equally well in our scenario. The gap in representation size can be exponential; for example, the TDD representing the identity tensor is linear in the number of qubits, whereas the explicit tensor has exponentially many entries. The main objective of this paper is to find a good contraction plan when the tensors are represented as TDDs; in particular, we aim for small intermediate TDD representations.

Example 9. *To reduce the complexity, we now simplify our running example to use only unary gates. Figure 6 shows the TDDs for the tensor network consisting of a Z gate followed by two H gates. Clearly, the two H gates cancel each other out, and thus, the circuit is equivalent to a Z gate.*

Initially, we can choose between two contractions: (1) of the Z and the first H gate or (2) of the first and the second H gate. The results are shown in Figure 6d and Figure 6e, respectively. As can be seen, the first contraction has a smaller intermediate representation. Thus, this contraction would be preferred for us. The final TDD is equivalent to the TDD of the Z gate, as expected.

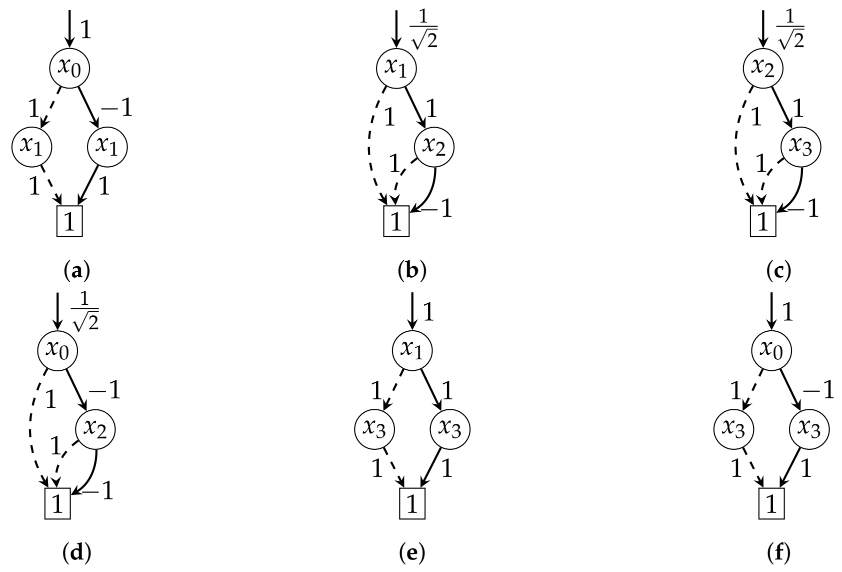


Figure 6. TDDs for the tensor network $\overset{x_0}{-}Z\overset{x_1}{-}H_1\overset{x_2}{-}H_2\overset{x_3}{-}$. (**a**) TDD for $\overset{x_0}{-}Z\overset{x_1}{-}$; (**b**) TDD for $\overset{x_1}{-}H_1\overset{x_2}{-}$; (**c**) TDD for $\overset{x_2}{-}H_2\overset{x_3}{-}$; (**d**) the resulting TDD when contracting $\overset{x_0}{-}Z\overset{x_1}{-}H_1\overset{x_2}{-}$ via x_1; (**e**) the resulting TDD when contracting $\overset{x_1}{-}H_1\overset{x_2}{-}H_2\overset{x_3}{-}$ via x_2; (**f**) the (unique) resulting TDD when contracting the whole tensor network.

In the simplest case, a (offline) contraction heuristic works in two phases. In the *planning phase*, a contraction plan is found. In the *execution phase*, the plan is executed. In this paper, we focus on (online) heuristics which alternately perform one planning step (i.e., identify the next contraction step) and one execution step (i.e., execute the contraction step). The advantage of such an online heuristic is that it does not have to predict what happens after the next step.

3. Contraction Heuristics for TDD Networks

In this section, we present two contraction heuristics for TDD networks. Both heuristics work *online*, i.e., they interleave planning and execution by only determining the next contraction step and executing it immediately.

The first heuristic, which we call the *lookahead method*, is a greedy method which results in a good contraction order but requires additional computations in the planning steps. Motivated by this success, the second heuristic, which we call the *counting method*, approximates the first heuristic; as a result, it still produces an overall good contraction order but also uses fast planning steps.

3.1. Lookahead Method

Our first heuristic method follows a greedy policy. In the planning step, each possible next contraction of two connected TDDs is evaluated, and the contraction with the smallest resulting TDD is selected for the next execution step. Consequently, we call this heuristic the *lookahead method*.

The planning steps of the lookahead method seem computationally expensive. In particular, there are concerns regarding the cost of the (hypothetical) contractions in a single planning step and that these computations have to be repeated in each planning step. Fortunately, there are a number of reasons why the method still works favorably in practice. Below we argue that the number of contractions is limited, most of the contractions take place in the first planning step, where computations are cheap, and only very few updates are required in each of the later planning steps.

Let m be the number of TDDs in the initial TDD network. First, while there are $\mathcal{O}(m^2)$ pairs of TDDs to contract in the worst case, tensor networks obtained from quantum circuits, which typically only contain unary and binary gates, are sparsely connected. Hence, the number of connected pairs is $\leq 4\ m$, and the quadratic worst case is not observed in our scenario. Second, only the very first planning step requires many contractions. After any execution step of contracting two TDDs D_1 and D_2, all other hypothetical contraction results from the previous planning step that involve neither D_1 nor D_2 remain valid. Hence, by storing the previous contraction results, planning steps in later iterations involve only a few new computations. Third, contractions in the first planning step are often efficient, since they correspond to the combination of two quantum gates. Even writing out the matrix representation (a 4×4 matrix in the worst case) is cheap.

We summarize the lookahead method in Algorithm 1. The algorithm maintains a priority queue (Q), which stores pairs (k, v) of keys k (natural numbers) and values v. The values are pairs of TDDs, while the keys are the size of the TDD resulting from their contraction. Note that the queue can store multiple values for the same key. Moreover, note that a priority queue allows for efficient `enqueue` and `remove` operations ($\mathcal{O}(\log \ell)$ with ℓ elements in the queue), and an $\mathcal{O}(1)$ `remove_minimum` operation. First, we initialize Q with each possible pair of TDDs for contraction (which can be easily obtained from the layout of the quantum circuit), where we store the resulting size if we were to contract these TDDs.

In each iteration of the contraction loop (line 6), we start with a planning step that selects two TDDs D_1 and D_2 whose contraction result is minimal. In case of multiple options (line 7), any tie breaker can be used (we use the implementation of Q). In the execution step, we execute the contraction to obtain the resulting TDD D_3. Finally, we need to update Q because TDDs D_1 and D_2 do not exist anymore, and instead, we establish connections with the new TDD D_3. Note that the loop in line 9 iterates over the whole queue, which takes $\mathcal{O}(\ell)$ steps (where ℓ is the number of elements in the queue).

Algorithm 1 Lookahead method

```
   Input  : TDD network
   Output: TDD describing the contraction result
   // initialize priority queue with possible contraction results
1  $Q \leftarrow$ create_priority_queue();
2  foreach connected TDDs $\{D_1, D_2\}$ do
3      $D_3 \leftarrow$ contract($\{D_1, D_2\}$);
4      enqueue($Q$, $|D_3|$, $\{D_1, D_2\}$);
5  end
6  while contractions left do
       // planning step part 1: dequeue minimal element from queue
7      $\{D_1, D_2\} \leftarrow$ remove_minimum($Q$);
       // execution step
8      $D_3 \leftarrow$ contract($\{D_1, D_2\}$);
       // planning step part 2: update queue for next iteration
9      foreach $D_{12}$ in $[D_1, D_2]$ do      // update entries with $D_1$ or $D_2$ to $D_3$
10         foreach $(k', \{D_{12}, D_4\}) \in Q$ do
11             remove($Q$, $k'$, $\{D_{12}, D_4\}$);
12             $D_5 \leftarrow$ contract$\{D_3, D_4\}$;
13             enqueue($Q$, $|D_5|$, $\{D_3, D_4\}$);
14         end
15     end
16 end
17 return resulting TDD
```

In our implementation, instead of only storing the size and then recomputing the contraction D_3 from $\{D_1, D_2\}$, we also directly store the TDD D_3 in the queue. This way, we avoid recomputing the contraction later. While this seems expensive in terms of memory consumption, the overall memory consumption is typically dominated by the most expensive (single) contraction results, which would already fail in the lookahead step before even getting to the contraction step.

We conclude the algorithm description with an example:

Example 10. *We run Algorithm 1 on the TDD network from Example 9 (Figure 6). Initially, the priority queue contains the following key-value pairs: $Q_0 = [3 \mapsto \{\{Z, H_1\}\}, 4 \mapsto \{\{H_1, H_2\}\}]$. In this case, the lookahead method will execute the contraction of Z and H_1 (as it results in the smaller TDD) to the TDD ZH_1. Since H_1 is part of both pairs in the queue, both pairs will be removed from the queue, and only the final contraction will be inserted: $Q_1 = [4 \mapsto \{\{ZH_1, H_2\}\}]$. Finally, this contraction is executed and the algorithm terminates.*

3.2. Counting Method

While we argued that the lookahead method is not as expensive as it seems at first glance, the planning time is still significant and dominates the total contraction time, specifically for circuits with many gates. In particular, the most expensive contractions still have to be evaluated but are then not actually applied.

When we evaluated the lookahead method, we observed a pattern that the next contraction would often happen between two TDDs that have not been contracted in a long time. Our intuition is that, in many cases, the resulting TDD is bigger than both input TDDs, and thus, one should often favor small TDDs.

Thus, to imitate a typical contraction order as obtained from the lookahead method, our second heuristic method is designed to avoid the redundant tentative contractions and simply rank TDDs based on counting how long they have not been involved in any contraction. Accordingly, we call our second heuristic method the *counting method*.

We summarize the lookahead method in Algorithm 2. Instead of a priority queue, the algorithm only requires a standard first-in-first-out (FIFO) queue (Q), for which `enqueue` and `dequeue` are $\mathcal{O}(1)$ operations. The order in which to initialize the queue is arbitrary (in our implementation, we select the order we obtain from iterating over the network).

Algorithm 2 Counting method

```
Input  : TDD network
Output : TDD describing the contraction result
// initialize queue with possible contraction pairs
1  $Q$ ← create_queue();
2  foreach connected TDDs $\{D_1, D_2\}$ do                    // arbitrary order
3      enqueue($Q$, $\{D_1, D_2\}$);
4  end
5  while contractions left do
       // planning step part 1: dequeue next element from queue
6      $\{D_1, D_2\}$ ← dequeue($Q$);
       // execution step
7      $D_3$ ← contract($\{D_1, D_2\}$);
       // planning step part 2: update queue for next iteration
8      foreach $D_{12}$ in $[D_1, D_2]$ do          // update entries with $D_1$ or $D_2$ to $D_3$
9          foreach $\{D_{12}, D_4\} \in Q$ do
10             remove($Q$, $\{D_{12}, D_4\}$);
11             enqueue($Q$, $\{D_3, D_4\}$);
12         end
13     end
14 end
15 return resulting TDD
```

The first planning step simply dequeues the pair $\{D_1, D_2\}$ from Q that has been in the queue for the longest time. The execution step computes the contraction, resulting in TDD D_3. Finally, we need to move all other pairs involving the TDDs D_1 and D_2 to the end of the queue and update these entries to the new TDD D_3. We note that the corresponding loop in line 8 can be implemented efficiently (i.e., without iterating over the whole queue) with little overhead by implementing the queue as a doubly linked list and storing for each TDD D all pointers to pairs involving D.

We again conclude the algorithm description with the same example as before:

Example 11. *As in Example 10, we run Algorithm 2 on the TDD network from Example 9 (Figure 6). Initially, the queue Q contains the following TDD pairs (note that the order is arbitrary): $Q_0 = [\{Z, H_1\}, \{H_1, H_2\}]$. In this case, the counting method will execute the contraction of Z and H_1 (as it comes first in the queue) to the TDD ZH_1. Again, since H_1 is part of both pairs in the queue, both pairs will be removed from the queue, and only the final contraction will be inserted: $Q_1 = [\{ZH_1, H_2\}]$. Finally, this contraction is executed and the algorithm terminates.*

4. Evaluation

In this section, we report on the evaluation results of our contraction heuristics when applied to checking the equivalence of quantum circuits. First, we describe our implementation and the benchmark problems we use in the evaluation. Next, we compare our heuristics to other state-of-the-art contraction heuristics implemented in the tool COTENGRA [15]. Finally, we compare our approach to checking the equivalence of quantum circuits to the tool QCEC [24], which has the same aim.

4.1. Experimental Setup

Our implementation consisted of three components. First, we implemented a TDD library in C++ for which we substantially extended a preliminary implementation (we extended the preliminary implementation from https://github.com/Veriqc/TDD_C, accessed on 25 May 2024). Second, we implemented our contraction heuristics in C++. Third, we implemented our equivalence checking approach in Python for easy integration with COTENGRA (see below).

All experiments were run on an Intel i5-14600KF CPU with 27 GB RAM and a GeForce RTX 4070 Super 12 GB GPU. The operating system was Ubuntu 20.04, and we used C++17 and Python 3.9.

To evaluate our approach, we use the application to check equivalence of quantum circuits. For this purpose, we choose quantum circuits from the MQT BENCH [33] benchmark suite, which contains many well-known quantum circuits on four different levels of a compilation pipeline. In our evaluation, we select the *algorithmic* (first) level and the *target-dependent* (third) level, which differ significantly in their gate sets and circuit layouts, making the task of equivalence checking challenging.

In particular, we use the following quantum circuits (short explanations are available online (https://www.cda.cit.tum.de/mqtbench/benchmark_description, accessed on 1 November 2024): Deutsch-Jozsa algorithm (DJ), Greenberger-Horne-Zeilinger state preparation (GHZ), graph state preparation (GS), quantum Fourier transformation applied to entangled qubits (QFTE), real amplitudes ansatz with random parameters (RAR), and W state preparation (WS). All circuits have a parametric number of qubits and are available with varying sizes. This allows us to study the scalability in the number of qubits.

4.2. Evaluation: Lookahead and Counting Heuristics

In the first experiment, we evaluate our two proposed heuristics. As discussed before, we should expect that the lookahead heuristic spends significantly time on the contraction planning because it tries out all possible next contractions before selecting the best one. The counting heuristic was motivated because it circumvents this cost.

Figure 7 shows that this intuition is indeed correct. The planning time of the counting heuristic is significantly shorter compared to the lookahead heuristic.

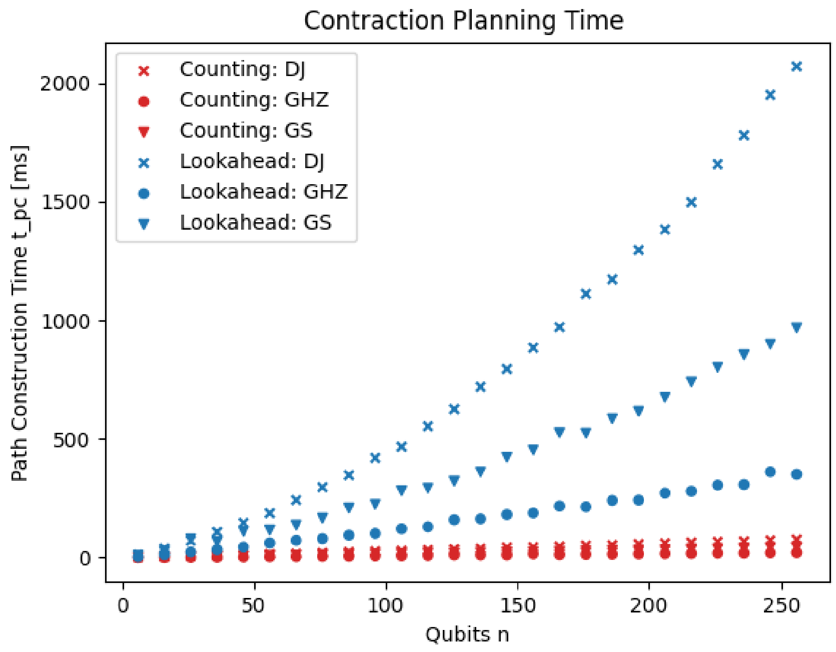


Figure 7. Contraction planning times of the lookahead heuristic and the counting heuristic. The times (y-axis) are shown for three different families of quantum circuits (different markers) and for varying numbers of qubits (x-axis).

Figure 8 shows the total time for equivalence checking with both heuristics. The counting heuristic clearly outperforms the lookahead heuristic (note the log scale). Consequently, we mainly use the counting heuristic in later experiments.

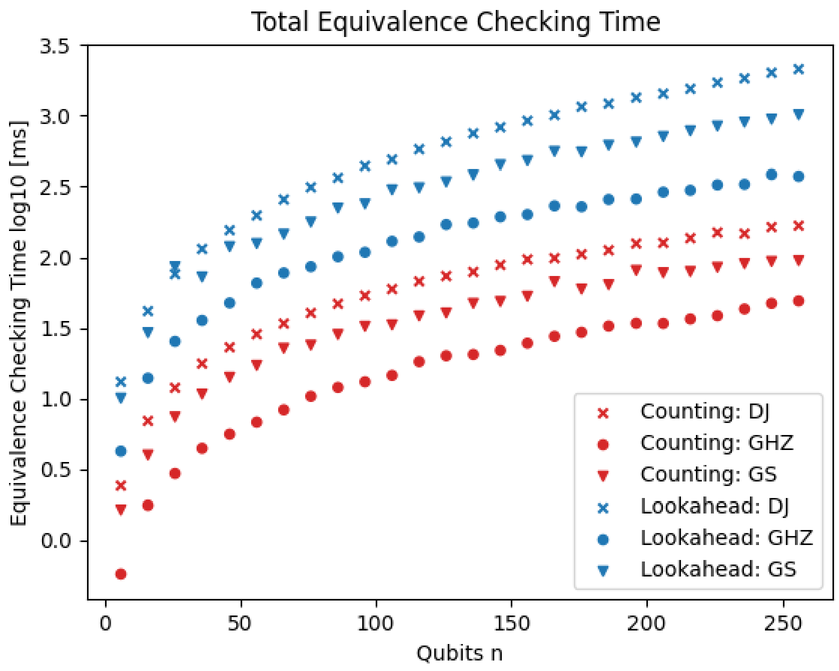


Figure 8. Total equivalence checking time using the lookahead and the counting heuristics. The times (y-axis; log-scale) are shown for three different families of quantum circuits (different markers) and for varying numbers of qubits (x-axis).

4.3. Evaluation: COTENGRA *Contraction Heuristics*

In this experiment, we compare our counting heuristic to other contraction heuristics from COTENGRA [15], a state-of-the-art contraction planning tool for tensor networks implemented in Python. Based on prior experimentation, we select the two heuristics that perform best for our setup: *Betweenness* and *RandomGreedy*. The Betweenness heuristic is based on edge betweenness centrality (a graph-theoretic concept) and identifies graph communities in order to perform contractions in these communities first. The Random-Greedy heuristic samples random contraction plans multiple times and selects the best plan according to the sum of the expected amount of floating-point operations of the entire plan.

To gain better insights into how these different heuristics work, we visualize the contraction order for the DJ circuit in Figure 9. The lookahead heuristic indeed spreads the contractions over the whole network more or less uniformly. As before, the counting heuristic imitates this behavior, although in a different order. The *Betweenness* heuristic starts on the very top of the network and sweeps to the bottom, while the *RandomGreedy* heuristic uses yet another order.

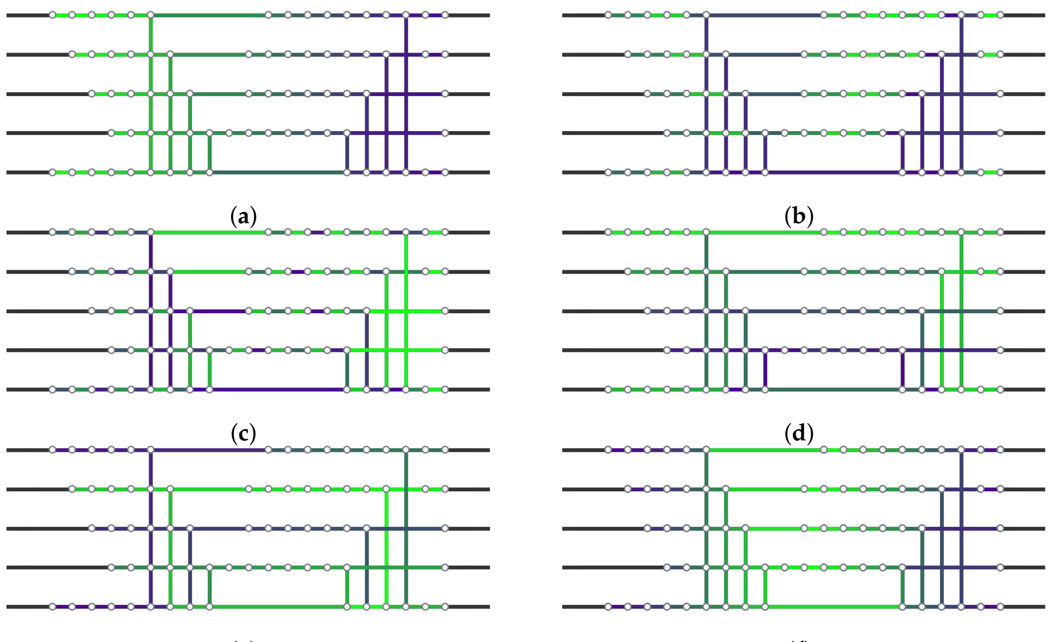
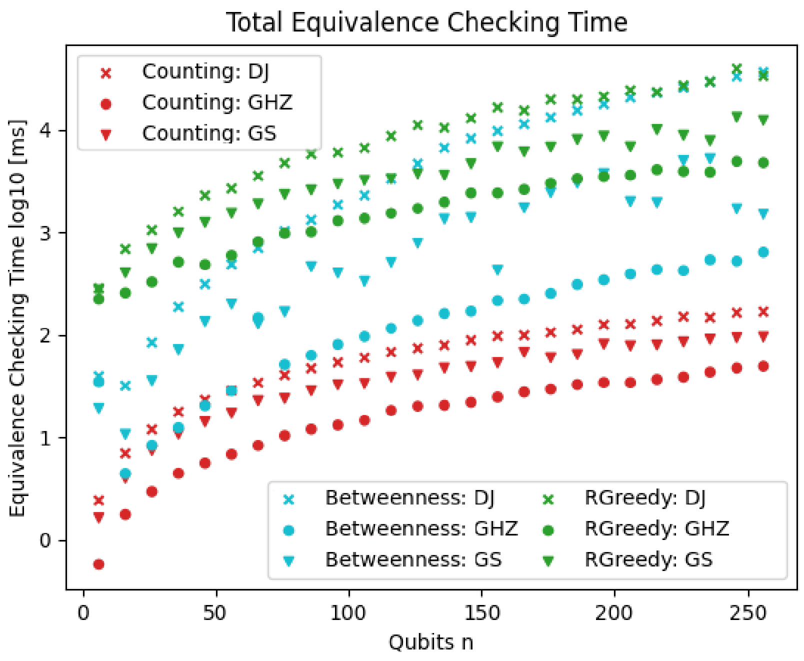


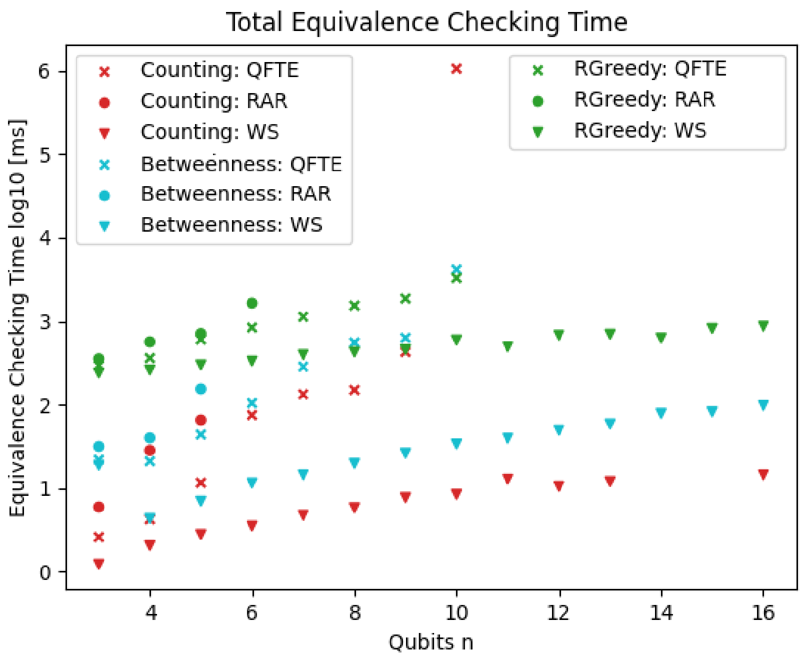
Figure 9. Visualization of different contraction orders on the DJ circuit. The contraction order is encoded in the color spectrum from green (contracted first) to purple (contracted last). (**a**) A simple contraction from left to right; (**b**–**e**) our two heuristics and the two other heuristics implemented in COTENGRA, specifically: (**b**) lookahead heuristic; (**c**) counting heuristic; (**d**) COTENGRA's Betweenness heuristic; (**e**) COTENGRA's RandomGreedy heuristic; (**f**) the heuristic used in [25], which exploits the fact that the tensor network originates from two circuits and contracts corresponding gates proportionally from the inside out.

Figure 10 shows the results of the three evaluated heuristics for our application: the equivalence checking of quantum circuits. The results of the counting heuristic are identical to those presented in Figure 8. Of the two COTENGRA heuristics, Betweenness generally performs better than RandomGreedy.

Our counting heuristic consistently outperforms the COTENGRA heuristics on all three circuits by 1–2 orders of magnitude. This is surprising, as the COTENGRA heuristics are much more sophisticated. We conjecture that the advantage of the counting heuristic stems from two sources: the very efficient planning time and the specific application of quantum circuit equivalence. Studying the performance of our heuristics on further tensor network contraction benchmarks beyond our application is outside the scope of this paper.



(**a**)



(**b**)

Figure 10. Equivalence checking time using our TDD-based approach with the counting heuristic and the heuristics implemented in COTENGRA. The times (y-axis; log-scale) are shown for six different families of quantum circuits (different markers) and for varying numbers of qubits (x-axis). (**a**) Three families of quantum circuits: DJ, GHZ, GS; (**b**) Three families of quantum circuits: QFTE, RAR, WS.

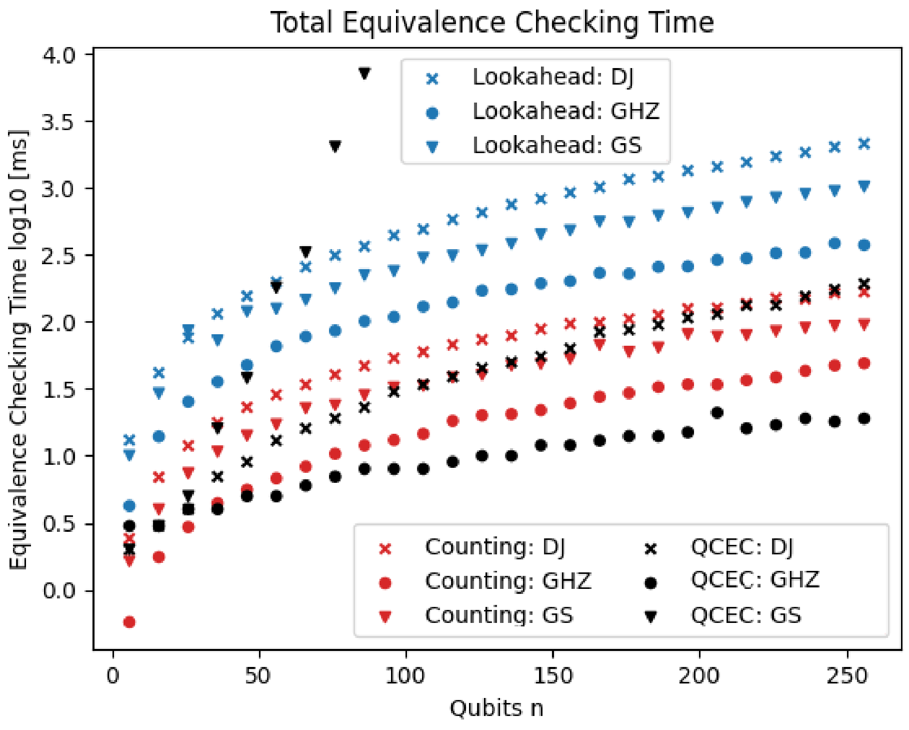
4.4. Evaluation: Equivalence Checking of Quantum Circuits

As another baseline for our equivalence checking approach, we use QCEC (Quantum Circuit Equivalence Checking) [24]. QCEC is a C++ library and implements equivalence checking based on quantum decision diagrams (QDDs), which is similar in nature to our approach and thus makes for a good comparison [23]. QCEC also uses ZX-calculus [27] in a preprocessing step, as it can sometimes very quickly prove circuit equivalence. Due to its fundamental difference and orthogonal performance, we deactivate the usage of ZX-calculus in the evaluation for a fair comparison to the QDD approach.

As outlined in Section 2.4, to check for the equivalence of two circuits C_1 and C_2, we ask whether the combined circuit $C_1\,C_2^\dagger$ is the identity. We then transform the combined circuit into a tensor network and represent the tensors as TDDs, which we then contract using a contraction heuristic. Finally, we compare the result to the TDD for the identity matrix.

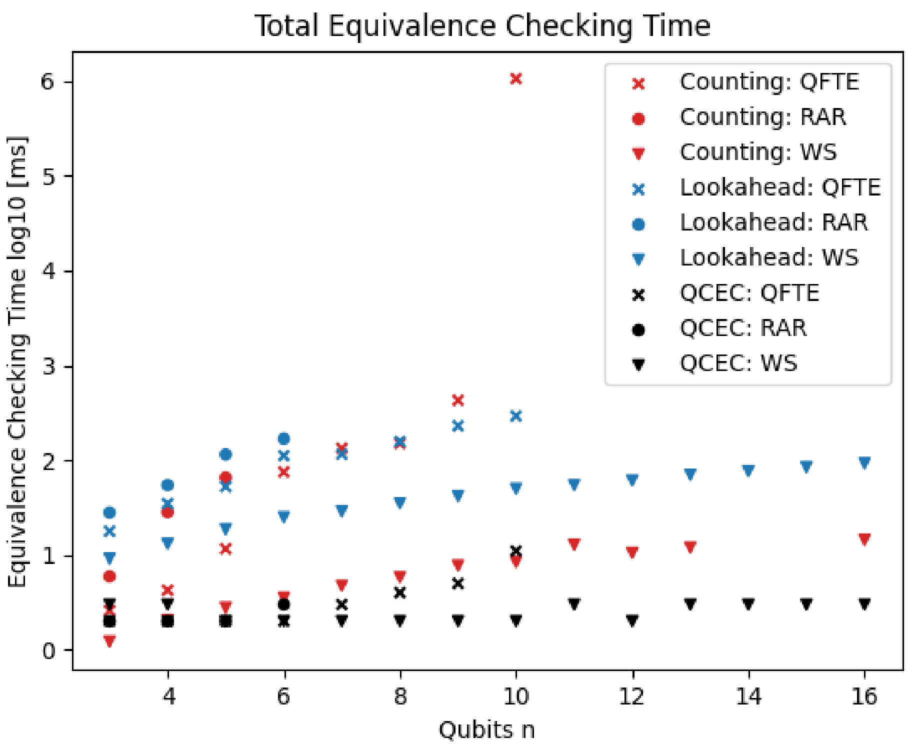
QCEC's strategy is related but differs in several aspects. First, it operates at the gate level and represents gates as QDDs. Second, it instead considers the circuit $C_1\,I\,C_2^\dagger$ with an additional identity gate in the middle; then, it explores the circuit from the inside out, alternately multiplying one gate to the left or right in each step (see Figure 9f). Similar to our approach, the resulting QDD is finally compared to the QDD for the identity matrix.

Figure 11 shows the results of the comparison of our approach with QCEC. For the first three quantum circuits (first plot), there is no clearly best approach, but the results are mostly consistent within the same family of circuits. QCEC performs better on the GHZ circuit. For the DJ circuit, QCEC is faster for small instances, but the run time grows faster than for the counting heuristic, which consequently outperforms on the largest instances. For the GS circuit, QCEC only manages to get up to 96 qubits within a time limit of 1000 s, while our approach scale much better, easily reaching 256 qubits. On the second plot (some data points in Figure 11b (QFTE beyond 10 qubits; RAR beyond 6 qubits; WS for 14 or 15 qubits; and the counting heuristic) are missing due to issues with floating-point precision; this exemplifies that quantum computations are nontrivial to implement on a classical computer), QCEC outperforms our approach on all circuits. Notably, the lookahead heuristic performs better than the counting heuristic on the QFTE circuit.



(**a**)

Figure 11. *Cont.*



(**b**)

Figure 11. Equivalence checking time using our TDD-based approach with the counting heuristic and the QDD-based approach implemented in QCEC. The times (y-axis; log-scale) are shown for six different families of quantum circuits (different markers) and for varying numbers of qubits (x-axis). (**a**) Three families of quantum circuits: DJ, GHZ, GS; (**b**) three families of quantum circuits: QFTE, RAR, WS.

5. Conclusions

In this paper, we have described a framework for checking the equivalence of quantum circuits. In our framework, we reduce this problem to the contraction of a tensor network. We represent the tensor network efficiently using tensor decision diagrams (TDDs). As our main contribution, we have described the first contraction heuristics specifically designed for TDD networks. Our first heuristic, called the lookahead method, evaluates all possible next contractions with their effect on the TDDs and, in the end, picks the best contraction greedily. While the lookahead method performs very well, it also comes with a computationally heavy overhead. Our second heuristic, called the counting method, is designed to imitate the lookahead method while avoiding its overhead by distributing the contractions over the entire network.

In our experimental evaluation, we have demonstrated that our contraction heuristics outperform those implemented in the state-of-the-art library COTENGRA on this application. Moreover, we have demonstrated that equivalence checking via TDD networks is a viable approach that can sometimes outperform the QDD-based approach QCEC.

There are several directions for future work. First, it would be interesting to study our contraction heuristics on other contraction benchmarks beyond quantum circuit equivalence, which we left open. Second, by studying the benchmarks where QCEC performs better than our TDD-based approach, we hope to identify more powerful contraction heuristics. Third, our current implementation is purely sequential and does not exploit parallelization; modern hardware supports massive concurrent operations, and since our contraction heuristics operate in different regions of the TDD network, the computations would be easily parallelizable. Finally, we see great potential in finding contraction plans with the help of modern machine learning.

Author Contributions: Conceptualization, all authors; methodology, all authors; software, C.B.L. and S.B.O.; validation, C.B.L. and S.B.O.; formal analysis, C.B.L. and S.B.O.; investigation, C.B.L. and S.B.O.; resources, C.B.L. and S.B.O.; data curation, C.B.L. and S.B.O.; writing—original draft preparation, C.S.; writing—review and editing, C.B.L., S.B.O., K.G.L. and C.S.; visualization, C.B.L., S.B.O. and C.S.; supervision, K.G.L. and C.S.; project administration, C.B.L., S.B.O., K.G.L. and C.S.; funding acquisition, K.G.L. and C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Independent Research Fund Denmark under reference number 10.46540/3120-00041B, DIREC—Digital Research Centre Denmark under reference number 9142-0001B, and the Villum Investigator Grant S4OS under reference number 37819.

Institutional Review Board Statement: Not applicable.

Data Availability Statement: Our implementation is openly available at https://github.com/Simonbolsen/P10 and https://github.com/ChBLA/TDDLinux (accessed on 1 November 2024).

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Harrow, A.W.; Montanaro, A. Quantum computational supremacy. *Nature* **2017**, *549*, 203–209. [CrossRef]
2. Bauer-Marquart, F.; Leue, S.; Schilling, C. symQV: Automated symbolic verification of quantum programs. In *Lecture Notes in Computer Science, Proceedings of the 25th International Symposium, FM 2023, Lübeck, Germany, 6 March 2023*; Springer: Cham, Switzerland, 2023; Volume 14000, pp. 181–198. [CrossRef]
3. Chen, Y.; Chung, K.; Lengál, O.; Lin, J.; Tsai, W. AutoQ: An automata-based quantum circuit verifier. In *Lecture Notes in Computer Science, Proceedings of the Computer Aided Verification, CAV 2023, Paris, France, 17–22 July 2023*; Springer: Cham, Switzerland, 2023; Volume 13966, pp. 139–153. [CrossRef]
4. Zhang, Q.; Saligane, M.; Kim, H.; Blaauw, D.T.; Tzimpragos, G.; Sylvester, D. Quantum circuit simulation with fast tensor decision diagram. In Proceedings of the 25th International Symposium on Quality Electronic Design (ISQED), San Francisco, CA, USA, 3–5 April 2024; IEEE: New York, NY, USA, 2024; pp. 1–8. [CrossRef]
5. Jiménez-Pastor, A.; Larsen, K.G.; Tribastone, M.; Tschaikowski, M. Forward and Backward Constrained Bisimulations for Quantum Circuits. In *Lecture Notes in Computer Science, Proceedings of the TACAS, Luxembourg, 6–11 April 2024*; Springer: Cham, Switzerland, 2024; Volume 14571, pp. 343–362. [CrossRef]
6. Alarcon, S.L.; Elster, A.C. Quantum Computing and High-Performance Computing: Compilation Stack Similarities. *Comput. Sci. Eng.* **2022**, *24*, 66–71. [CrossRef]
7. Shaik, I.; van de Pol, J. Optimal Layout Synthesis for Quantum Circuits as Classical Planning. In Proceedings of the ICCAD, San Francisco, CA, USA, 28 October–2 November 2023; IEEE: New York, NY, USA, 2023, pp. 1–9. [CrossRef]
8. Javadi-Abhari, A.; Treinish, M.; Krsulich, K.; Wood, C.J.; Lishman, J.; Gacon, J.; Martiel, S.; Nation, P.D.; Bishop, L.S.; Cross, A.W.; et al. Quantum computing with Qiskit. *arXiv* **2024**, arXiv:2405.08810. [CrossRef]
9. Janzing, D.; Wocjan, P.; Beth, T. "Non-identity-check" is QMA-complete. *Int. J. Quantum Inf.* **2005**, *3*, 463–473. [CrossRef]
10. Ji, Z.; Wu, X. Non-Identity Check Remains QMA-Complete for Short Circuits. *arXiv* **2009**, arXiv:0906.5416. [CrossRef]
11. Tanaka, Y. Exact non-identity check is NQP-complete. *Int. J. Quantum Inf.* **2010**, *8*, 807–819. [CrossRef]
12. Orús, R. Tensor networks for complex quantum systems. *Nat. Rev. Phys.* **2019**, *1*, 538–550. [CrossRef]
13. Meirom, E.A.; Maron, H.; Mannor, S.; Chechik, G. Optimizing Tensor Network Contraction Using Reinforcement Learning. In Proceedings of the ICML, Baltimore, MA, USA, 17–23 July 2022; PMLR: New York, NY, USA, 2022; Volume 162, pp. 15278–15292.
14. Wahl, T.B.; Strelchuk, S. Simulating Quantum Circuits Using Efficient Tensor Network Contraction Algorithms with Subexponential Upper Bound. *Phys. Rev. Lett.* **2023**, *131*, 180601. [CrossRef] [PubMed]
15. Gray, J.; Kourtis, S. Hyper-optimized tensor network contraction. *Quantum* **2021**, *5*, 410. [CrossRef]
16. Viamontes, G.F.; Markov, I.L.; Hayes, J.P. Improving Gate-Level Simulation of Quantum Circuits. *Quantum Inf. Process.* **2003**, *2*, 347–380. [CrossRef]
17. Abdollahi, A.; Pedram, M. Analysis and synthesis of quantum circuits by using quantum decision diagrams. In Proceedings of the DATE, Munich, Germany, 6–10 March 2006; European Design and Automation Association: Leuven, Belgium, 2006; pp. 317–322. [CrossRef]
18. Miller, D.M.; Thornton, M.A. QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In Proceedings of the ISMVL, Singapore, 17–20 May 2006; IEEE Computer Society: New York, NY, USA, 2006; p. 30. [CrossRef]
19. Niemann, P.; Wille, R.; Miller, D.M.; Thornton, M.A.; Drechsler, R. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2016**, *35*, 86–99. [CrossRef]
20. Vinkhuijzen, L.; Coopmans, T.; Elkouss, D.; Dunjko, V.; Laarman, A. LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States. *Quantum* **2023**, *7*, 1108. [CrossRef]

21. Hong, X.; Zhou, X.; Li, S.; Feng, Y.; Ying, M. A Tensor Network based Decision Diagram for Representation of Quantum Circuits. *ACM Trans. Design Autom. Electr. Syst.* **2022**, *27*, 60:1–60:30. [CrossRef]
22. Viamontes, G.F.; Markov, I.L.; Hayes, J.P. Checking equivalence of quantum circuits and states. In Proceedings of the ICCAD, San Jose, CA, USA, 4–8 November 2007; IEEE Computer Society: New York, NY, USA, 2007; pp. 69–74. [CrossRef]
23. Burgholzer, L.; Wille, R. Advanced Equivalence Checking for Quantum Circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2021**, *40*, 1810–1824. [CrossRef]
24. Burgholzer, L.; Wille, R. QCEC: A JKQ tool for quantum circuit equivalence checking. *Softw. Impacts* **2021**, *7*, 100051. [CrossRef]
25. Burgholzer, L.; Ploier, A.; Wille, R. Simulation Paths for Quantum Circuit Simulation With Decision Diagrams What to Learn From Tensor Networks, and What Not. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2023**, *42*, 1113–1122. [CrossRef]
26. Hong, X.; Feng, Y.; Li, S.; Ying, M. Equivalence Checking of Dynamic Quantum Circuits. In Proceedings of the ICCAD, San Diego, CA, USA, 29 October–3 November 2022; ACM: New York, NY, USA, 2022; pp. 127:1–127:8. [CrossRef]
27. Coecke, B.; Duncan, R. Interacting Quantum Observables. In *Lecture Notes in Computer Science, Proceedings of the ICALP, Reykjavik, Iceland, 7–11 July 2008*; Springer: Cham, Switzerland, 2008; Volume 5126, pp. 298–310. [CrossRef]
28. Peham, T.; Burgholzer, L.; Wille, R. Equivalence Checking of Quantum Circuits with the ZX-Calculus. *IEEE J. Emerg. Sel. Topics Circuits Syst.* **2022**, *12*, 662–675. [CrossRef]
29. Thanos, D.; Coopmans, T.; Laarman, A. Fast Equivalence Checking of Quantum Circuits of Clifford Gates. In *Lecture Notes in Computer Science, Proceedings of the ATVA, Singapore, 24–27 October 2023*; Springer: Cham, Switzerland, 2023; Volume 14216, pp. 199–216. [CrossRef]
30. Hong, X.; Ying, M.; Feng, Y.; Zhou, X.; Li, S. Approximate Equivalence Checking of Noisy Quantum Circuits. In Proceedings of the DAC, San Francisco, CA, USA, 5–9 December 2021; IEEE: New York, NY, USA, 2021, pp. 637–642. [CrossRef]
31. Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information*, 10th ed.; Cambridge University Press: Cambridge, UK, 2016.
32. Markov, I.L.; Shi, Y. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* **2008**, *38*, 963–981. [CrossRef]
33. Quetschlich, N.; Burgholzer, L.; Wille, R. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* **2023**, *7*, 14. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.