Summary

Quantum circuit equivalence checking is a central problem in the quantum computing field. As each quantum computer uses different gate sets and logical to physical qubit mappings, quantum circuits must be compiled individually for each such quantum computer architecture. The initial logical quantum circuits may be transformed drastically by compilation. While these transformations should preserve the functionality of initial circuit, checking it is non-trivial. Verifying the equivalence of circuits before and after completion is, therefore, crucial to utilise quantum computing resources effectively.

Quantum circuits and their gates rely on mathematical representations when simulated on a classical computer. Each gate is defined as a complex-valued unitary matrix. The matrices can be combined to represent entire circuits, which is a central part of most quantum computations performed on classical computers, including quantum circuit equivalence checking. As the size of the matrix representation of a circuit is exponential in the number of qubits, using matrices is infeasible for even moderately sized quantum circuits.

From the classical domain, it is well-known that decision diagram structures can for some problems avoid an exponential blow-up. Similarly, for the quantum domain, decision diagrams allow for a more efficient representation of both quantum gates and circuits by exploiting redundancies. The Tensor Decision Diagram is one such data structure, often allowing for a more compact representation than its matrix counterpart. Tensor Decision Diagrams are based on tensors which generalise matrices and allow connected gates to be combined in any order. Combining tensors or Tensor Decision Diagrams is called contraction.

The order gates are contracted in, called the contraction plan, has a big impact on the sizes of intermediate representations. Finding better contraction plans both makes equivalence checking faster and also makes it possible to contract bigger and more complicated circuits. Some heuristics already exist for creating contraction plans. However, these heuristics only consider the tensor representations of gates and do not factor in the more efficient representation of the Tensor Decision Diagram.

To include the Tensor Decision Diagrams in a contraction plan heuristic, this thesis considers a neural network model that learns to predict the sizes of Tensor Decision Diagrams. For this, a dataset is made from contracting on random circuits with existing heuristics. Different partitions of the dataset are made to train variations of the model with different biases.

The neural network model is used in two different heuristics: a heuristic made to plan and contract in discrete windows and a heuristic based on Monte Carlo tree search. The tree search heuristics is limited by the ability to pick the best plan between several. We find that having discrete windows, which allow the actual sizes of intermediate products to be used, has no significant effect on the performance of the heuristic compared to making the entire plan before contraction.

Two handcrafted heuristics are made. The Lookahead heuristic does all the available contractions and continuously proceeds with the one that yields the smallest result until the entire circuit has been contracted. The EMIT heuristic is designed so that new TDDs are not immediately used for other contractions. It is found that EMIT is the best performing of the considered heuristic.

When comparing to existing heuristics it is found that our model-based and handcrafted heuristics are faster. The EMIT heuristic combined with a new C++ implementation has vastly improved the equivalence checking method and made it competitive compared to alternative equivalence checking tools that utilise different decision diagram structures.

Contraction Heuristics using Tensor Decision Diagram Size Estimation for Quantum Circuit Equivalence Checking

Christian Bøgh Larsen, Simon Brun Olsen
{cbla, solse}19@student.aau.dk

Department of Computer Science, Aalborg University

***Abstract*—Quantum circuit equivalence checking inherently depends on how quantum circuits are represented on a classical computer. Tensor Decision Diagrams (TDDs) efficiently represent quantum gates and quantum circuits. TDDs allow the equivalence checking process to be done in multiple possible orderings, called contraction plans. Current contraction plan heuristics do not consider the sizes of TDDs but only the tensors they are based on. This leaves unexplored potential in the heuristics for constructing efficient contraction plans. This unexplored potential is the focus of this thesis.**

A way of estimating the size of a TDD resulting from a contraction is devised to create new heuristics. To predict such sizes, we train a neural network model on a custom dataset. The prediction model is then used to create two different heuristics: a heuristic that plans and contracts in discrete windows and a heuristic based on Monte Carlo tree search.

Two handcrafted heuristics are also evaluated. The first, Lookahead, does the actual contractions as opposed to estimating them. The second, called EMIT, emulates the Lookahead heuristic by contracting the TDDs that have been contracted the least.

All the contraction planning heuristics are evaluated on benchmark circuits. We conclude that using the neural network model has advantages over existing heuristics on select circuits. The Lookahead heuristic outperforms existing heuristics as it requires no initial planning time. The EMIT heuristic is the fastest overall on most circuits as it neither has an elaborate planning step nor additional expensive contractions. We outperform state-of-the-art contraction planning tools on equivalence checking using TDDs, and we achieve results comparable in time with state-of-the-art equivalence checking tools.

***Index Terms*—Quantum Circuits, Tensor Decision Diagram, Neural Network, Contraction Plan**

I. Introduction

Quantum computation has seen a rapid increase in interest from both the scientific community as well as the industrial area. Quantum computers promise to speed up certain types of algorithms to a potentially exponential degree [4]. While quantum computers are currently impractical for industrial use, the rate of improvement is similar to that of classical computers, and it is expected that in the near future, practically usable quantum computers will exist. To ensure that there are ample tools and methods to support quantum computation once the hardware matures, there is a need to adapt methods from the classical domain or develop new methods and tools specifically designed for the quantum domain.

A central task in quantum computing is designing and improving quantum algorithms, also called quantum circuits. A related task is to compile quantum circuits [20] between different abstraction levels [6] to bridge the gap between design and implementation on a specific quantum computer. Compilation algorithms may also perform optimisations in the circuits, such as removing redundant gates or using a specific qubit ordering to reduce the use of SWAP gates [21]. Since compilation to lower abstraction levels also depend on the hardware architecture, including the allowed gates, the resulting compiled algorithm may vary greatly both in what gates it is composed of and how many [19]. As such, when developing compilation methods, it is imperative to ascertain that the functionality of an algorithm is not changed. Checking quantum circuit equivalence is used to verify that two circuits are functionally equivalent even if they differ significantly in their appearance [3].

For a quantum circuit, being functionally equivalent means that for any input state, the two circuits should agree on the resulting output state. While quantum circuit equivalence checking is as simple as checking whether two matrices are equal, the practicality of such an approach is limited as the sizes of matrix representations grow exponentially with the number of qubits in a quantum circuit. As such, there is a need for more efficient methods for verifying the equivalence of two circuits [13]. While being exponential in the worst case, performing equivalence checking within practical time limits may still be possible.

This thesis is based on a previous work by the same authors [15]. The previous work investigated the feasibility of combining a quantum circuit equivalence checking method by Burgholzer et al. [3] with the decision diagram structure, called Tensor Decision Diagrams (TDDs), defined by Hong et al. [12]. The equivalence checking method used contraction plans describing how to combine representations of quantum gates. TDDs represent quantum gates and can be contracted together to represent entire circuits. It was shown that equivalence checking may be performed using contraction plans on TDDs, and several contraction plan heuristics were evaluated.

The previous work lacked contraction plan heuristics that could utilise information about the contracted TDDs. Such information may improve the method by allowing the data structure to be better exploited. This thesis further investigates the use of contraction planning heuristics to improve the equivalence checking method regarding scaling and time consumption.

A. Related Works

To reduce the resource requirements of representing quantum circuits, research has been conducted in adapting decision diagrams from the classical domain to the quantum domain. In the classical domain, decision diagrams have been applied to exponentially large expressions with great success. One decision diagram structure adapted to the quantum domain is the Quantum Decision Diagram, QDD [24]. QDDs have shown much potential in exploiting redundancies in the matrices and can often represent exponentially large matrices in polynomial space [1][24][12][3].

Another decision diagram structure applicable to quantum circuits is the Tensor Decision Diagram (TDD) structure [12], with several applications in quantum computation [10][11][9]. TDDs are designed to represent tensors in tensor networks. Since circuits can be interpreted as tensor networks, TDDs can be used to represent the gates of the circuits. Hong et al. [12] show that TDDs, while feasible for representing quantum circuits, are slower than QDDs when performing operations on them. However, as mentioned by Burgholzer et al. [2], the added flexibility of TDDs using tensor contraction as opposed to the matrix multiplication of QDDs, may allow for practically faster applications.

Another representation of circuits is ZX-calculus, which revolves around rewriting circuits such that redundancies are removed [5]. ZX-calculus has also been used for equivalence checking of quantum circuits [17] and has been implemented as part of the state-of-the-art Quantum Circuit Equivalence Checker from the Munich Quantum Toolkit (QCEC) [3]. QCEC also makes use of QDDs to do equivalence checking.

Burgholzer et al. [3] explore an equivalence checking setup from Viamontes et al. [22] in which two circuits are combined in such a manner that their combined representation is identity if they are equivalent. In such a setup, the order in which one combines the gates to reach the final representation of the entire circuit may affect the intermediate results. Using this setup, Burgholzer et al. investigate some simple heuristics to determine such orderings of combining gates. Burgholzer et al. use QDDs and find that exploiting the construction of the combined circuits produces the fastest contractions, as the intermediate products are often close to or equal to the identity. Burgholzer et al. [2] further investigate the effects of simulation paths for simulating quantum circuits using QDDs. Burgholzer et al. use a contraction planning tool meant for contractions in tensor networks. The setup is heavily constrained by the use of the QDD data structure. It is shown that using contraction plans improves the time consumption from the linear plans, but is worse than an optimal proportional heuristic.

Wahl et al. [23] investigate the power of contraction plans for simulating quantum circuits using tensor network representations. Using the topology of the quantum circuits, Wahl et al. achieve significant improvement in the simulation of quantum circuits compared to other well-known methods, and comparable to the plans provided by a state-of-the-art contraction planning tool cotengra [8].

Meirom et al. [16] propose to use a reinforcement learning (RL) approach to finding fast contraction plans for tensor network contractions. The gates are represented as tensors without any data structure for space optimisation. Meirom et al. formulate the tensor network contraction ordering problem as a Markov decision process, which opens up the possibility of an RL-based solution. The tensor network is interpreted as a graph, such that a graph neural network (GNN) model using message passing [25] can be applied. The GNN based on the input graph outputs a probability distribution over the edges. Using these probabilities, an edge is sampled to be contracted. Meirom et al. manage to outperform the state-of-the-art contraction planning tool cotengra using this approach to contraction planning. Meirom et al. also show that their method improves simulation on certain quantum circuits by up to four times.

B. Contributions

This thesis further investigates the effects of contraction plans with regards to contraction time for quantum circuit equivalence checking using TDDs from the previous work by the same authors [15]. Other contraction planning heuristics leverage information on either the topology of the circuits or the tensors of the tensor network representations of the circuits [3][2][23][8]. This thesis seeks to leverage the information of the intermediate TDDs resulting from performing equivalence checking. The previous work performed contractions in a proof-of-concept Python implementation with relatively poor performance. This thesis uses a C++ implementation of TDDs and their contractions.

The contributions of this thesis are thus: A faster C++ implementation of TDD contractions, which allows results to be comparable to state-of-the-art equivalence checking methods; A neural network model capable of predicting the size of a TDD resulting from a contraction; Several contraction planning heuristic leveraging predictions or actual sizes of TDDs to reduce the sizes of the intermediate results. Two heuristics use neural network models to predict the sizes of possible contractions. The first contracts in discrete windows allowing actual values to be used for subsequent planning steps, and the second is based on Monte Carlo tree search. Two other handcrafted heuristics are made. The first does all immediately available contractions before continuing with the smallest one, and the second orders contractions such that they are distributed evenly between the TDDs. Additionally, the thesis provides experiments showing the performance of the developed heuristics and comparison with the state-of-the-art tool QCEC.

II. Background

To understand and utilise the methods described in this thesis, some background knowledge is required. This includes some basics of quantum computation as well as the problem of quantum circuit equivalence checking. It also includes definitions relating to tensors and tensor networks. Since the purpose of this thesis is to facilitate different contraction planning heuristics, contraction planning is defined in relation to both tensor networks and TDDs. Two heuristics from cotengra are briefly introduced. Finally, Tensor Decision Diagrams, TDDs, are introduced formally and visual examples of the data structure are given.

A. Quantum Computation Basics

The basic unit of all quantum computation is the qubit. The qubit is to quantum computation as the bit is to classic computation. All operations in quantum computation revolve around transforming the states of qubits. While a bit represents a Boolean value using 0 and 1 for *false* and *true*, respectively, a qubit represents a linear combination of 0 and 1. A qubit may be either 0 or 1, referred to as basis states, or some combination of 0 and 1. When a qubit is in a linear combination of 0 and 1 and not a basis state, it is said to be in a state of superposition. Superposition may be collapsed by measurement. This means that the qubit becomes a classical bit and probabilistically assumes either the 0 or 1 basis state. The superposition of a qubit is defined by two amplitudes, one complex value for each basis state of the qubit. When the superposition collapses during measurement, the probability of the qubit to assume either basis state is determined by the amplitudes.

Assume a qubit $|q_0\rangle$ (using the Dirac/BraKet notation [7]) is in a superposition with amplitudes $\alpha, \beta \in \mathbb{C}$. Then, the qubit is described as:

$$|q_0\rangle = \alpha|0\rangle + \beta|1\rangle, \text{ such that } |\alpha|^2 + |\beta|^2 = 1 \tag{1}$$

Measuring $|q_0\rangle$ results in basis state $|0\rangle$ with probability $|\alpha|^2$ and results in basis state $|1\rangle$ with probability $|\beta|^2$.

A quantum state is the combined state of one or more qubits. Since the quantum state may be in a superposition of each combination of basis states, a quantum state has 2^n amplitudes for n qubits. A quantum state can also be represented as a state vector of its complex-valued amplitudes.

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

$$\alpha = [\alpha_{000}, \alpha_{001}, ..., \alpha_{111}]^\mathsf{T} \qquad |\alpha| = 2^n \tag{2}$$

which represent the linear combination of all the basis states:

$$|q_0q_1q_2\rangle = \alpha_{000}|000\rangle + \alpha_{001}|001\rangle + ... + \alpha_{111}|111\rangle \tag{3}$$

For this quantum state the probability of measuring $|000\rangle$ is $|\alpha_{000}|^2$.

To accommodate computation, quantum gates (henceforth gates), as well as measurements, are used to transform quantum states. Measurements transform quantum states into classical states, whereas gates transform quantum states into possibly different quantum states. Gates may be applied to any number of qubits. Each gate has an associated unitary matrix, describing the transformation of the state vector. Gates being described by unitary matrices means that the property of squared amplitudes equating to probabilities is preserved when gates are applied. It also means that the inverse of a gate is easily described as the conjugate transposed matrix. Compared to general matrices, it is always possible and potentially much faster to find the inverse of the matrix of a quantum gate.

Given a quantum state consisting of a single qubit q_0 in basis state $|0\rangle$, the Hadamard gate H transforms the state into a state of superposition. The matrix of the Hadamard gate is:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{4}$$

Example 2 (Quantum Gates)**.** *Using a quantum state of $|q_0\rangle = 1\cdot|0\rangle + 0\cdot|1\rangle$, the transformation of gate H on the state results in the following state vector:*

$$H|q_o\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\cdot\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} \tag{5}$$

and written as a quantum state:

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{6}$$

Since all quantum gates are represented by unitary matrices for which there exist inverse matrices completely reversing their transformations, no information is masked during quantum computation. This is unlike classical computation where operations are often not revertible, such as applying an *or* gate where the original input values are not determinable by the output of the gate.

While a single gate is sufficient to represent any quantum algorithm disregarding the use of measurement, it is hardly practical to manually construct unitary matrices to represent exponentially large state vector transformations. As such, an abstraction of a sequence of gates applied to some quantum state, a quantum circuit (henceforth circuit), is needed. On Figure 1 an example quantum circuit using three gates on two qubits can be seen. Since gates are defined by matrices, any sequence of gates corresponds to sequentially applying the matrix representation of all the gates to the initial state vector. A circuit is then just a description of some gates, which qubits they act on, and the sequence in which they must be applied.

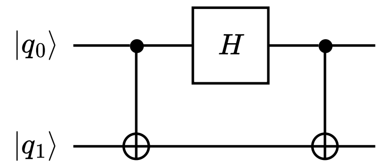

Fig. 1: Example of a quantum circuit consisting of two CNOT-gates and one Hadamard gate. The circuit transforms quantum states with two qubits.

Consider some circuit C defined by the following gates:

$$C = g_1, g_2, ..., g_{|C|} \tag{7}$$

The gates of the circuit can be represented by their matrices. Note that the order of the gates is reversed as matrix multiplication works from right to left:

$$U_C = U_{g_{|C|}} \cdot ... \cdot U_{g_2} \cdot U_{g_1} \tag{8}$$

The application of the circuit onto the initial state vector α is defined as (using matrix notation):

$$U_C \cdot \alpha = U_{g_{|C|}} \cdot ... \cdot U_{g_2} \cdot U_{g_1} \cdot \alpha \tag{9}$$

Thus the application of a circuit to a state is the sequential application of the gates to the state vector dictated by the sequence the gates appear in the circuit.

When applying a gate applicable to state vectors of n qubits to a state vector representing n' qubits with $2^{n'}$ amplitudes, where $n \neq n'$, an expansion on the state or the gate is required. For the expansion of states, ancillary qubits may be used to act only as intermediate results and discarded when done with

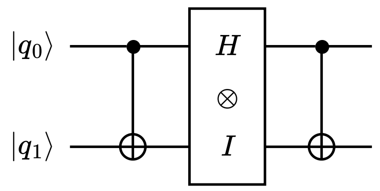

Fig. 2: The circuit resulting from expanding the Hadamard gate H using the tensor product on the circuit from Figure 1.

computation. To expand quantum gates, the tensor product between the gate matrices and identity matrices can be used. The tensor product is similarly used to combine quantum state vectors and parallel gates.

Example 3. *Consider the circuit in Figure 1. To combine the Hadamard gate with either of the CNOT-gates, it must be expanded. As such, an identity gate can be inserted in parallel with the Hadamard gate. To compose the identity gate and the Hadamard gate, the tensor product between the matrix representation of the two gates can be computed. The resulting circuit can be seen in Figure 2, and the matrix representation of the tensor product can be seen here:*

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{10}$$

Computing the result of applying a circuit to a state on a classical computer is referred to as quantum simulation, which can significantly speed up quantum circuit equivalence checking in the negative case while incurring only a small overhead to the overall checking [3].

B. Quantum Circuit Equivalence

The ability to verify that two quantum circuits are functionally equivalent, that is, for any input they yield the same output, is a necessity when developing new algorithms and optimisation schemes.

Since circuits are sequences of gates and matrices can represent gates, every circuit can be represented by one matrix capturing the combined effect of all constituent gates. Each circuit being represented by one matrix also indicates that it is possible to verify whether two circuits perform the same transformations, simply by verifying that the matrices representing the circuits are identical up to a global phase.

Definition 1 (Circuit Equivalence)**.** *Two circuits C with n qubits and C' with n' qubits and with matrix representation U_C and $U_{C'}$, respectively, are functionally equivalent, that is, $C = C'$, iff $n = n'$ and for some global phase $e^{i\theta}$:*

$$\exists\ \theta \in [0, 2\pi)\ .\ U_C = e^{i\theta} \cdot U_{C'} \tag{11}$$

Unless the two circuits are trivially identical, the matrices representing the circuits must be computed in their entireties before equivalence checking can be performed. Since computing these matrices is expensive, the invertibility of circuits and gates can be exploited.

Definition 2 (Circuit inverse)**.** *Given some circuit $C = c_1 \cdot c_2 \cdot \ldots \cdot c_{|C|}$, then there exists gates $c_1^{-1} \ldots c_{|C|}^{-1}$ such that $\forall_{i=1..|C|}(c_i \cdot c_i^{-1} = I)$. Then the inverse of circuit C is $C^{-1} = c_{|C|}^{-1} \cdot \ldots \cdot c_2^{-1} \cdot c_1^{-1}$. Observe that:*

$$\begin{aligned} C \cdot C^{-1} &= c_1 \cdot c_2 \cdot \ldots \cdot c_{|C|} \cdot c_{|C|}^{-1} \cdot \ldots \cdot c_2^{-1} \cdot c_1^{-1} \\ &= c_1 \cdot c_2 \cdot \ldots \cdot c_{|C|-1} \cdot c_{|C|-1}^{-1} \cdot \ldots \cdot c_2^{-1} \cdot c_1^{-1} \\ &= c_1 \cdot c_1^{-1} = I \end{aligned} \tag{12}$$

If two circuits C_1 and C_2 are equivalent such that $C_1 = C_2$, and since a circuit can be inverted as per Definition 2 such that $C_2 \cdot C_2^{-1} = I$, then clearly it follows that $C_1 \cdot C_2^{-1} = I$. This means that instead of having to check whether the matrices for the two circuits agree on all values, one can construct a combined circuit $C_1 \cdot C_2^{-1}$ and check whether this circuit is the identity. Constructing the combined circuit of one circuit with the inverse of the other circuit to check for identity is referred to as the combined circuit setup. Figure 3 shows an example of a combined circuit, where the left circuit C_1 is a single SWAP-gate and the right circuit C_2 is inverted into C_2^{-1}.

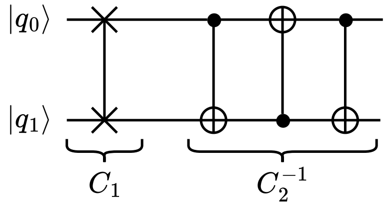

Fig. 3: Example of a combined circuit, where the left circuit is a single SWAP-gate and the right is the inverse of three CNOT-gates which also perform a SWAP operation.

While there is no immediate advantage in using the 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. By cleverly performing matrix multiplication on gates from both circuits such that the intermediate results become identity matrices, the final equivalence checking can be significantly sped up. Additionally, the combined circuit can with the use of other data structures also reduce the representation of intermediate results. Both of these effects are later observed.

C. Tensors

Tensors are another way to represent individual gates as a more general version of matrices. Tensors generalise matrices by having any number of dimensions rather than simply rows and columns. Each dimension has an index variable, resulting in an index vector. Each specific index vector thus corresponds to an element in the tensor. Figure 5 shows a tensor with its indices.

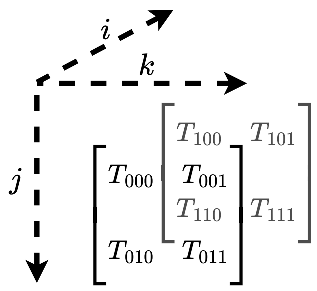

Fig. 5: Example of a three-dimensional tensor with corresponding indices i, j, and k.

Tensors can also represent quantum states. In this case, the tensors will have an index for each qubit in the quantum state. For the tensors used here, each index of the tensors has two possible values, one for each of the two basis states of a qubit. As both quantum states and the matrices of quantum gates consist of complex values, the elements of the tensors used here are complex values. Tensors are thus limited here to the form $T_{\overrightarrow{\mathrm{I}}} \in \mathbb{C}^{2^{|\overrightarrow{\mathrm{I}}|}}$, with $\overrightarrow{\mathrm{I}}$ being the vector of index variables. For vectors, lowercase symbols will be used to denote a specific vector of index values, for example, $\overrightarrow{\mathrm{i}} \in \{1, 2\}^{|\overrightarrow{\mathrm{I}}|}$.

Example 4. *Consider the tensor $T_{i,j,k}$ depicted in Figure 6. Then $T_{0,0,0}$ is a valid entry in $T_{i,j,k}$.*

Tensors are often depicted as either vectors when dealing with quantum states or matrices when the tensors represent gates. As shown in Figure 6 the placement of elements is determined by the indices. It can also be seen that rows and columns are interchangeable as they are purely illustrative and do not hold any meaning for the tensor.

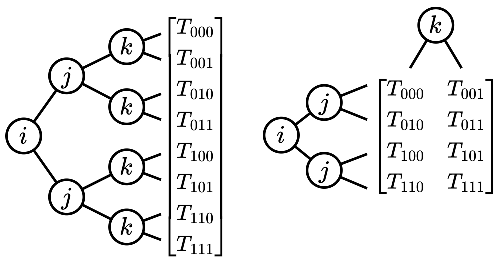

Fig. 6: A tensor depicted as a column vector and as a matrix showing how indices correspond to the position of the elements.

D. Tensor Networks

Just as gates can be combined into quantum circuits so can tensors be combined into tensor networks to represent those circuits. A tensor network is a graph structure where the vertices represent tensors. An edge in the tensor network connects tensors if they share an index. The same index can therefore only be shared by two tensors, and the words **edges** and **indices** are used interchangeably in the context of tensor networks. Since circuits contain inputs and outputs which are only partially connected until a state vector is supplied, the tensor network also has similar input and output indices. These input and output indices are then only connected to one tensor and are referred to as outer edges.

Definition 3 (Tensor Network)**.** *A tensor network is an undirected graph $G = (V, E, E_{outer}, S)$, where $V \subset \bigcup_{m \in \mathbb{N}} \mathbb{C}^{2^m}$ is the set of tensors that make up the vertices, S is the set of index variables, $E \subset V \times V \times 2^S$ is a set of inner edges and $E_{outer} \subset V \times 2^S$ is a set of outer edges. Each edge includes a set of index variables.*

Example 5 (Tensor Network)**.** *Consider a tensor network as illustrated in Figure 4 on the left. The tensor network is defined by the four-tuple $G_{ex} = (V, E, E_{outer} S)$. The tensor network G_{ex} initially consists of three tensors such that $V = \{ CX_{fgjk}, H_{gh}, CX_{hikl} \}$. The index set S of the tensor network is determined by the possible index variables of all tensors in the tensor network, here $S = \{ f, g, h, i, j, k, l \}$ as also seen by the edges in the figure. The input and output edges, here called outer edges E_{outer}, of a tensor network are the edges that only attach to one tensor, that is, $E_{outer} \subset V \times 2^S$. The inner edges are then the remaining edges that attach to two tensors, such that $E_{inner} \subset V \times V \times 2^S$. For the tensor network in question:*

$$E_{outer} = \{ (CX_{fgjk}, \{f\}), (CX_{fgjk}, \{j\}), (CX_{hikl}, \{i\}), (CX_{hikl}, \{l\}) \} \tag{13}$$

$$E_{inner} = \{ (CX_{fgjk}, H_{gh}, \{g\}), (CX_{fgjk}, CX_{hikl}, \{k\}), (H_{gh}, CX_{hikl}, \{h\}) \} \tag{14}$$

To simplify the tensor networks, multiple edges between two tensors are combined into a single edge. This is possible since an edge represents two tensors sharing some indices, and having two such edges simply means that the tensors share the union of indices of the two edges. This simplification is assumed for all tensor networks going forward.

Similarly to how the gate matrices can be combined into a matrix representation of a circuit, the tensors of the tensor

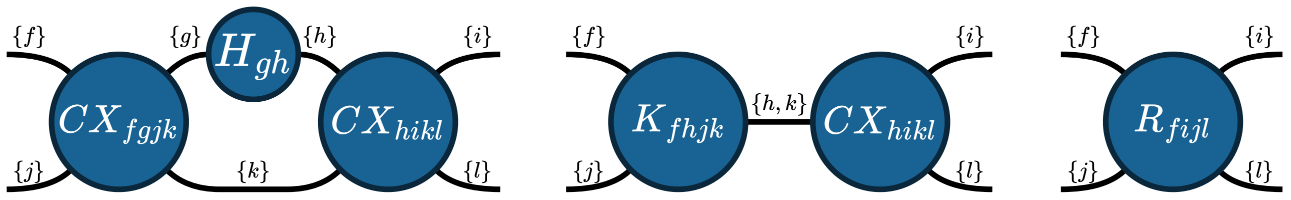

Fig. 4: The tensor network (left) of the small circuit in Figure 1 being contracted into a single tensor (right). The edges between the tensors are labelled with the sets of indices shared by the tensors.

network can be combined into a single tensor representing the same circuit. The operation used for combining tensors is contraction which generalises matrix multiplication. The operation multiplies the elements of each tensor with corresponding index values together while summing out shared indices.

Definition 4 (Tensor Contraction)**.** *Let $S_{\overrightarrow{\mathrm{I}},\overrightarrow{\mathrm{H}}}$ and $T_{\overrightarrow{\mathrm{J}},\overrightarrow{\mathrm{H}}}$ be tensors with their index vectors split into their shared index variables in the vector $\overrightarrow{\mathrm{H}}$ and their unique index variables in the vectors $\overrightarrow{\mathrm{I}}$ and $\overrightarrow{\mathrm{J}}$, respectively. The combined tensor is denoted $R_{\overrightarrow{\mathrm{I}},\overrightarrow{\mathrm{J}}}$.*

$$R_{\overrightarrow{\mathrm{i}},\overrightarrow{\mathrm{j}}} = \sum_{\overrightarrow{\mathrm{x}} \in \{1,2\}^{|\overrightarrow{\mathrm{H}}|}} S_{\overrightarrow{\mathrm{i}},\overrightarrow{\mathrm{x}}} \cdot T_{\overrightarrow{\mathrm{j}},\overrightarrow{\mathrm{x}}} \tag{15}$$

Example 6 (Tensor Contraction)**.** *The first contraction illustrated in Figure 4 computes the tensor K_{fhjk} from the tensors CX_{fgjk} and H_{gh}.*

$$K_{fhjk} = \sum_{x \in \{1,2\}} CX_{fxjk} \cdot H_{xh} \tag{16}$$

Example 7 (Tensor Contraction)**.** *The second contraction illustrated in Figure 4 computes the tensor R_{fijl} from the tensors K_{fhjk} and CX_{hikl}.*

$$R_{fijl} = \sum_{x,y \in \{1,2\}} K_{fxjy} \cdot CX_{xiyl} \tag{17}$$

E. Contraction Planning

To use tensor networks for quantum circuit equivalence checking, it is possible to construct a tensor network based on the combined circuit setup as seen in Definition 3. The resulting tensor network thus represents identity if and only if the two circuits initially used to construct the combined circuit are equivalent. To check whether the tensor network represents the identity matrix, all edges excluding outer edges can be contracted. The tensor network resulting from contracting all inner edges should thus be a tensor representing the identity with n input and n output indices, where n is the number of qubits in the combined circuit.

A contraction plan describes in which order tensors should be contracted until only a single tensor representing the entire tensor network is left. While tensors sharing no indices may be contracted, the height of the result of a contraction is determined by the number of unique indices in the two tensors. For this reason, as well as simplifying what constitutes a valid contraction plan, only tensors with shared indices may be contracted according to some contraction plan. Since tensors are now limited to sharing indices to be candidates for contraction, a contraction plan may instead be described by edges. As such, each step of a contraction plan indicates an edge, and thereby some shared indices between two tensors, which must be contracted. The resulting graph then removes the contracted edge, and the two tensors that are contracted are substituted with the result of their contraction.

Considering the graph definition of tensor networks in Definition 3, a contraction plan or contraction path (used interchangeably) can be described formally as:

Definition 5 (Contraction Plan)**.** *A contraction plan Π on an initial tensor network $G^{(0)} = (V^{(0)}, E^{(0)}, E_{outer}, S)$ is an ordered list of steps $\pi_i \in \Pi$ for $1 \leq i \leq |\Pi|$, $\pi_i \in E^{(i)}$. Performing the last step in the contraction plan $\pi_{|\Pi|}$ results in the final graph $G^{(|\Pi|)} = (V^{(|\Pi|)}, E^{(|\Pi|)}, S)$, which consists of a single tensor representing the entire initial tensor network. As such, $|V^{(|\Pi|)}| = 1$ and $E^{(|\Pi|)} = \emptyset$. A contraction plan on a tensor network $G = (V, E, E_{outer}, S)$ has a length of $|\Pi| = |V| - 1$.*

Example 8 (Contraction Planning)**.** *Consider the tensor network used in Example 5. Figure 4 shows a valid contraction plan for the tensor network. Since the tensor network initially consists of three tensors, the length of a plan on this tensor network is always $|\Pi| = |V| - 1 = 2$. The plan shown in Figure 4 is: $\Pi = [(CX_{fgjk}, H_{gh}, \{g\}), (K_{fhjk}, CX_{hikl}, \{h, k\})]$, which is one of three valid plans. The first step of the plan is $\pi_1 = (CX_{fgjk}, H_{gh}, \{g\})$. The result of the first contraction is a new tensor K_{fhjk} as described by Definition 4. The updated graph $G^{(1)} = (V^{(1)}, E^{(1)}, S)$. The graph is updated to encapsulate the contraction such that $V^{(1)} = \{\ CX_{hikl}, K_{fhjk}\ \}$ and $E^{(1)} = \{\ (K_{fhjk}, CX_{hikl}, \{h, k\})\ \}$. Performing the second step of the plan $pi_2 = (K_{fhjk}, CX_{hikl}, \{h, k\})$ results in a new graph $G^{(2)} = (V^{(2)}, E^{(2)}, E_{outer}, S)$ with a single tensor such that $V^{(2)} = \{\ R_{fijl}\ \}$ and no remaining inner edges such that $E^{(2)} = \emptyset$.*

Since each step in the contraction plan mutates the graph, a mechanism for properly updating the graph to reflect the changes of the contraction is required. A contraction combines two tensors into a new one. The edges attached to the two contracted tensors that are not involved in the contraction must be changed such that they attach to the new tensor instead. The edge on which the contraction occurs based on can simply be removed.

For the edges, another consideration must be taken. Consider the case where two tensors v_1 and v_2 in tensor network $G = (V, E, S)$ each have an edge connected to some tensor v_{shared}, such that $(v_1, v_{shared}, J_1) \in E$ and $(v_2, v_{shared}, J_2) \in E$. Then when the two tensors are contracted over edge (v_1, v_2, J) resulting in tensor v, each of the edges connected to v_1 and v_2 should be updated to use v such that $(v_1, v_{shared}, J_1) \longrightarrow (v, v_{shared}, J_1)$ and $(v_2, v_{shared}, J_2) \longrightarrow (v, v_{shared}, J_2)$. Since tensor networks assume that at most one edge exists between any pair of tensors, the combination of updated edges connecting the same tensors must occur as part of the graph update.

Definition 6 (Graph Update)**.** *A contraction $\pi_i = (v_1, v_2, J)$ with shared indices $J \in 2^S$ such that $(v_1, v_2, J) \in E^{(i-1)}$ performed on the graph $G^{(i-1)}$ produces a new graph $G^{(i)} = (V^{(i)}, E^{(i)}, S)$, where $V^{(i)} = \{v\} \cup V^{(i-1)} \setminus \{v_1, v_2\}$ such that v is the result of performing tensor contraction on tensors v_1, v_2. Edges are updated according to Definition 7.*

Definition 7 (Graph Edges Update)**.** *Using Definition 6, the updated graph is $G^{(i)} = (V^{(i)}, E^{(i)}, S)$. A vertex v is part of an edge e, if: $\exists_{v' \in V, J \subset S}\ .\ e = (v, v', J) \vee e = (v', v, J)$. A set of indices J is part of an edge if there exist vertices v' and v'', such that $e = (v', v'', J) \in E$. The updated edges are given by:*

$$E^{(i-1)}_{old} = \{\ e \mid e \in E^{(i-1)} \wedge (v_1 \in e \vee v_2 \in e)\ \} \tag{18}$$

$$\begin{aligned} E^{(i-1)}_{upd} = \ \{\ (v, v_t, J) \mid e \in E^{(i-1)}_{old}, J \in e, \\ v_t \in e,\ v_t \notin \{v_1, v_2\}\ \} \end{aligned} \tag{19}$$

$$\begin{aligned} E^{(i-1)}_{new} = \{\ (v_k, v_m, J) \mid e \in E^{(i-1)}_{upd},\ v_k, v_m \in e, \\ v_k \neq v_m,\ J = \{\ \iota \mid \epsilon \in E^{(i-1)}_{upd},\ \epsilon = (v_k, v_m, \iota)\ \}\ \} \end{aligned} \tag{20}$$

$$E^{(i)} = E^{(i-1)}_{new} \cup E^{(i-1)} \setminus E^{(i-1)}_{old} \tag{21}$$

Example 9 (Graph Update)**.** *Consider again the tensor network used in Example 5 and depicted in Figure 4, with the first contraction step π_0 as used in Example 8. After performing the contraction of $(CX_{fgjk}, H_{gh}, \{g\})$, a new graph $G^{(1)} = (V^{(1)}, E^{(1)}, S)$, where the edges $E^{(1)}$ are updated according to Definition 7:*

$$\begin{aligned} E^{(0)}_{old} = \{\ (CX_{fgjk}, H_{gh}, \{g\}), \\ (CX_{fgjk}, CX_{hikl}, \{k\}), (H_{gh}, CX_{hikl}, \{h\})\ \} \end{aligned} \tag{22}$$

$$E^{(0)}_{upd} = \ \{\ (K_{fhjk}, CX_{hikl}, \{k\}), (K_{fhjk}, CX_{hikl}, \{h\})\ \} \tag{23}$$

$$E^{(0)}_{new} = \{\ (K_{fhjk}, CX_{hikl}, \{h, k\})\ \} \tag{24}$$

Since $E^{(0)} = E^{(0)}_{old}$:

$$E^{(1)} = \{\ (K_{fhjk}, CX_{hikl}, \{h, k\})\ \} \tag{25}$$

For any contraction plan, it is important to note that the length of the plan is different from the initial number of edges. This is a consequence of the graph update step where several new edges connected to the same vertices are created. These edges are combined by taking the union of their indices. After updating the graph there will always be at most one edge between any pair of vertices in the graph.

F. Contraction Heuristics

Using tensors instead of matrices to represent gates and circuits gives many more options in which order to combine them. In the general case, it is NP-hard to find the optimal contraction plan [14]. Heuristics are therefore employed to find contraction plans within a reasonable time frame.

Cotengra [8] is a tool that provides contraction plan heuristics intended for use in tensor networks. Of the provided heuristics **Random Greedy** and **Betweenness** are found to perform best within the context of the article. **Random Greedy** samples multiple times and selects the best plan according to the sum of the expected amount of floating point operations of the entire plan. **Betweenness** is deterministic and uses graph communities to resolve the contractions sharing many indices first.

G. Tensor Decision Diagrams

When combining gate representations into a single representation of an entire circuit using either matrices or tensors, the amount of space required is exponential in the number of qubits in the circuit. It is therefore infeasible to handle even modestly sized circuits on a classical computer. To make this possible, redundancies in those representations can be exploited to make more efficient data structures.

For tensors one such data structure is a Tensor Decision Diagram, also called a TDD.

Definition 8 (Tensor Decision Diagram, adapted from [12])**.** *A Tensor Decision Diagram, is a directed, rooted, acyclic graph $F = (W, \Xi, index, w_g)$ over a set of indices S. The graph consists of a set of nodes W, a set of edges Ξ, a global weight $w_g \in \mathbb{C}$ for the incoming edge to the root node, and the function index, which are defined as follows:*

- *$W = W_N \cup \{w_T\}$, where W_N is a set of non-terminal nodes, and v_T is the single terminal node of the graph.*
- *$\Xi \subseteq W_N \times W \times \mathbb{C} \times \{\ low, high\ \}$. Edges therefore point from a non-terminal node to a node, it has a complex weight, and is either a low or high edge. Each node in W_N has exactly one outgoing low edge and one outgoing high edge.*
- *The function index : $W_N \longrightarrow S$ assigns an index from the index set to each non-terminal node.*

All TDDs are assumed to be ordered, normalised, and reduced, as defined in [12]. TDDs being ordered means that indices have a predefined total order, and all non-terminal nodes in a TDD therefore have a partial order based on the indices that they are mapped to by the *index* function. TDDs being normalised means that low edges are limited to values of 0 and 1, and all edges with a 0 weight point to the terminal node W_T. TDDs being reduced means that there are no duplicated nodes. From these properties, it follows that the TDDs are canonical, meaning that there is a unique TDD for every tensor and index ordering.

Figure 7 shows the TDDs of the tensors CX_{fgjk}, H_{gh}, and K_{fhjk}. The TDD of K_{fhjk} is the result of the contraction between the other two TDDs, which represent the controlled not gate and the Hadamard gate respectively.

The size of a TDD is the number of nodes it contains including the terminal node. For the TDDs depicted in Figure 7, the sizes are from left to right: 8, 3, and 7, respectively.

III. Method

The following sections introduce a neural network model for predicting the size of TDDs and different ways to utilise such a model to create contraction plans. This includes contraction heuristics that work before contraction begins and heuristics that are used to find plans during contraction. After that, two handcrafted heuristics are introduced, one which computes actual contraction and greedily chooses the smallest available, and another which distributes contractions throughout the tensor network.

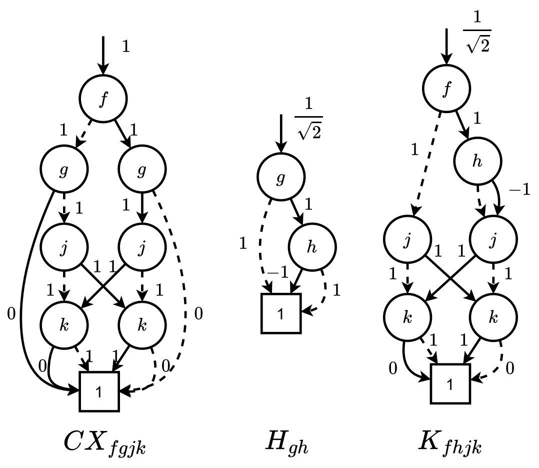

Fig. 7: The TDDs in the first contraction in Figure 4.

A. TDD Prediction Model

TDDs reduce the required space during the contraction process. In some cases, including for the identity, the space required is linear as opposed to exponential in the number of qubits. It is nevertheless difficult to fully utilise this potential to reduce representations, as the sizes of the TDDs are not known, and cannot be trivially computed when constructing contraction plans. As of writing this thesis, only performing the contraction itself allows for computing the size of the resulting TDD.

A feed-forward neural network model can be constructed to predict the size of a TDD resulting from the contraction of other TDDs. With a model predicting the resulting size of a contraction, one can perform more informed choices when choosing between contractions. Such a model can thus be used to make contraction plans that may better utilise the TDDs potential to reduce the required space. The model takes information from two TDDs and predicts the size of the TDD that is created after they have been contracted.

As TDDs cannot be directly passed to the neural networks, metadata is extracted in the form of floating-point numbers. For the model introduced here, the information given as input to the model from the two input TDDs is:

- The sizes of the constituent TDDs.
- The number of indices of the constituent TDDs.
- The number of gates, of each type in a fixed set of gates the constituent TDDs are constructed from.
- The number of shared indices between the two constituent TDDs.

The model uses a set of 13 gate types. Hence there are 15 floating point numbers given for each TDD and a single number given for the shared indices. Therefore, there are three different input sources, as shown on the left side of Figure 8. The total size of the input to the model is thus 31 values.

The model is designed such that the ordering of the two TDDs is inconsequential since TDD contraction is commutative. To achieve this the vectors from the TDDs go through the same linear layer L_{in}. The number of shared indices goes through the

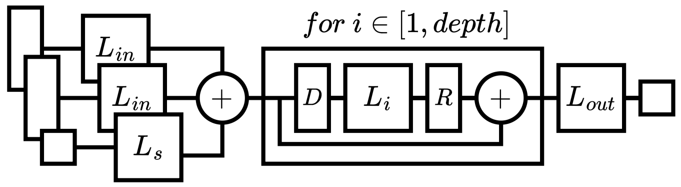

Fig. 8: The structure of the TDD size prediction model.

linear layer L_s, and finally, all three embeddings are summed to one vector embedding.

The main body of the model consists of a block that is repeated multiple times, determined by a hyperparameter $depth$. The block has a skip connection that adds the initial embedding vector of each iteration to the final embedding. The block then has a dropout layer D, with an associated hyperparameter for the dropout rate. Each iteration uses a different linear layer L_i followed by a non-linear activation function $ReLU$. The final step of the model is the linear layer L_{out}, which turns the embedding vector into the single-valued size prediction.

Definition 9 (TDD Prediction model)**.** *The TDD prediction model is a function that takes two TDDs from the vertices V of a tensor network and some amount of shared indices. The function returns a real-valued log_2 prediction of the size of the TDD given by contracting the two input TDDs.*

$$tpm: \ V \times V \times \mathbb{N} \to \mathbb{R}^+ \tag{26}$$

	$log_2(s)$	$\lvert J\rvert$	H	$CNOT$	...	T
CX_{fgjk}	3	4	0	1	...	0
H_{gh}	1.58	2	1	0	...	0
CX_{hikl}	3	4	0	1	...	0
K_{fhjk}	2.81	4	1	1	...	0
R_{fijl}	3	4	1	2	...	0

Table 1: Examples of the data extracted from TDDs. s is the size of the TDD, $|J|$ is the number of indices in the TDD, and gates, here with the actual gate names, indicate the number of each of the 13 previously mentioned gates that are in the TDDs.

Example 10 (TDD Prediction model input)**.** *Table 1 shows the information given to the model for the TDDs of the tensors in Figure 4 and Figure 7.*

B. Greedy Prediction Heuristic

The simplest way to utilise the prediction model is to use it in a greedy heuristic.

Definition 10 (Greedy Prediction Heuristic)**.** *Given an initial tensor network $G = (V^{(0)}, E^{(0)}, E_{outer}, S)$ and a TDD prediction model tpm, the steps of the plan $\pi_i \in \Pi$ is given as:*

$$\pi_i = \underset{(v_1,v_2,J)\in E^{(i)}}{\operatorname{argmin}} \ tpm(v_1, v_2, |J|) \tag{27}$$

This heuristic can be used in various ways depending on whether predictions are based on actual contracted TDDs or previous predictions. It is shown in Example 10 how it is possible to use size predictions as input to the model. The model uses the sizes, the number of indices, and the number of each gate type for each TDD. For index sets J and J' the index set of the new TDD is $(J \cup J') \setminus (J \cap J')$, as this is the side effect of Definition 7 from Section II-E. As such, the number of gates can be added together. Therefore only the sizes of the TDDs depend on the contraction and the predicted value can be used instead.

C. *Windowed-N Heuristic*

The Windowed-N heuristic uses the greedy prediction heuristic from Definition 10 and interleaves planning and contraction in discrete and equally sized segments called windows. Instead of always planning all steps, each window is planned fully before all the steps of the window are contracted. This process of planning and contracting a window is repeated until the entire circuit has been contracted. The Greedy Prediction heuristic from Section III-B may also be described as a Windowed-N heuristic but with a window size of the length of the entire plan. As such, the Greedy Prediction heuristic is referred to as Windowed-Max henceforth.

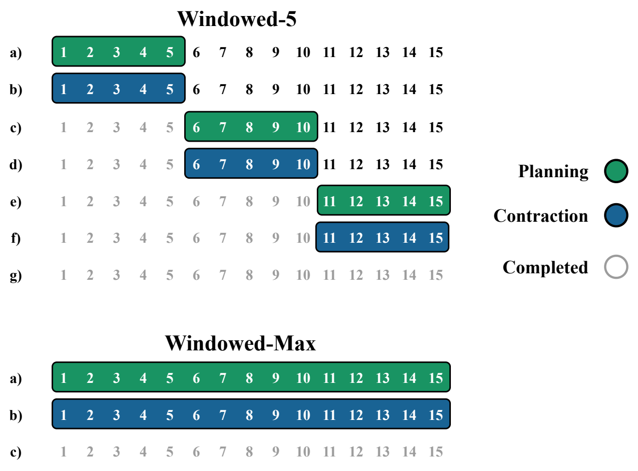
Windowed-5
a) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
c) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
d) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
e) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
f) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
g) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Planning
Contraction
Completed
Windowed-Max
a) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
b) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
c) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Fig. 9: Two different heuristics, Windowed-5 and Windowed-Max, applied to some tensor network with the entire plan being of length 15. The green box signifies planning, the blue box signifies contraction, and the greyed-out numbers indicate the completed steps of the contraction plan.

Figure 9 shows the general approach to the Windowed-N heuristic, here Windowed-5 and Windowed-Max, respectively. The heuristics in Figure 9 are applied to some circuits for which the entire plan is 15 steps. For the Windowed-5 heuristic, it can be seen that it initially plans five steps. Then, it contracts these five planned steps, updates the tensor network graph with the actual sizes as opposed to the predicted sizes, and then plans the next five steps. This continues until all steps are planned and contracted. For Windowed-Max, the same occurs, but since the window size is the total length of the plan, there is always only one window.

By adjusting the window size it is possible to ascertain whether the Windowed-N heuristic becomes prone to enter a destructive feedback loop for certain window sizes. A destructive feedback loop is where slightly wrong predictions cause even more wrong future predictions until the final prediction of the resulting TDD is completely different from its actual size.

D. *Tree Search*

The greedy contraction plans are made for the planning to be as fast as possible, while still using the model, by always making the best immediate choice. While the greedy approach may be fast, having heuristics that focus on finding better plans using more time is also relevant. One way to look for better plans is to consider multiple contraction plans made with some stochastic variant of the greedy prediction heuristic. While the planning time for such an approach increases compared to the greedy prediction heuristic, it may enable equivalence checking for more difficult circuits. The tree search approach utilises the developed neural network model for prediction, which allows it to make full plans before contraction begins. Thus, as in the Windowed-Max heuristic, all predictions made by the neural network model are based on the predictions made in the previous step.

The tree search contraction planning heuristic is based on Monte Carlo tree search. The idea is to look at the search tree of possible contractions and have it be expanded by consecutive contraction plans. The search tree is thus made up of nodes representing tensor networks that have undergone some amount of contractions. Outgoing edges from a node in the search tree are thus the contractions that are possible on the tensor network represented by that node.

Traversing the search tree from the root to a leaf is called a sample. The final plan is then the best-performing sample. Multiple samples can be explored by introducing randomness into the choice of which contraction is done at each step of the sample. The randomness is introduced by giving each possible contraction a weight in a probability distribution based on available information. Such information may include the immediate prediction for that contraction, the largest predicted size in previous samples using that edge, and the sum of predicted sizes seen on previous samples through that edge. This is done by repeatedly planning to the end and then collected information is back-propagated through the tree. Because each sample functions as a contraction plan, the tree search can be given a set amount of planning time and then return the best sample once the time is up.

By decorating the tree with information from the end of the plan, the idea is to use this information in the subsequent samples. Each sample would place information on the nodes it traverses in the tree to adjust the probability of including those nodes in future samples. The probability is determined by a weight function which gives a positive real number w_e for each possible contraction over an edge e. The probability is then given by normalising the weights of the considered contractions as shown in Equation 28.

$$P(e, E^{(i)}) = \frac{w_e}{\sum_{e' \in E^{(i)}} w_{e'}} \tag{28}$$

The simplest and only weight function considered here is the greedy weight function. Similarly to the greedy prediction contraction plan, it uses the immediate expected size of the contraction. It is used as a negative exponent which functions as a softmax, so that larger predicted sizes result in smaller weights.

Definition 11 (Greedy Weight Function)**.** *Based on the edge e from the set of edges E, a TDD prediction model tpm, and a parameter $\alpha \in \mathbb{R}^+$.*

$$\begin{gathered} w(e, tpm) = \alpha^{-tpm(v_1, v_2, |J|)} \\ where\ e = (v_1, v_2, J) \end{gathered} \tag{29}$$

Finally, once all samples have been computed one of them is chosen as the contraction plan based on a sample metric. The two sample metrics considered are **max** and **sum** which aggregate over the predicted sizes of the contractions of the plan.

Definition 12 (Sample Metric Max and Sum)**.** *Given a set of samples Z the sample metrics **sum** and **max** choose a sample Π_{sum} and Π_{max} respectively as contraction plans.*

$$\Pi_{sum} = \underset{\Pi_i \in Z}{\operatorname{argmin}} \sum_{(v_1, v_2, J) \in \Pi_i} tpm(v_1, v_2, |J|) \tag{30}$$

$$\Pi_{max} = \underset{\Pi_i \in Z}{\operatorname{argmin}} \max_{(v_1, v_2, J) \in \Pi_i} tpm(v_1, v_2, |J|) \tag{31}$$

E. Lookahead Contraction Planning Heuristic

The Windowed heuristic suffers mainly from inaccurate predictions resulting in worse plans. To resolve some of the issues regarding this, a heuristic can be designed that substitutes the prediction of the resulting size with the actual result by performing the contraction. As such, each step of the plan is determined in a greedy fashion by which actual contraction yields the smallest resulting TDD.

F. EMIT Heuristic

The Excitable Medium Interpretation of Tensor Network Contraction (EMIT) heuristic is inspired by physical excitable media; in particular, the refractory time. Refractory time refers to a period of time wherein a physical object does not react to stimuli, meaning that repeated events cannot occur at a rate faster than the refractory time. To apply such an idea to tensor network contractions, each tensor in the tensor network may be seen as having its own refractory time wherein it cannot be contracted.

The effect of such a refractory time for tensors is that the contractions are spread throughout the tensor network as opposed to being bunched up. Assuming that the size of TDDs grows faster than linear with the number of contractions, spreading out contractions between TDDs would reduce the overall sizes compared to repeatedly contracting the same TDDs.

The EMIT heuristic needs to make sure that after two TDDs have been contracted, the resulting TDD is not used again until all other TDDs have been involved in some contraction. In practice, this can be done by putting all tensor network edges into a queue and contracting over one edge at a time. The remaining edges that are attached to the resulting TDD are then moved to the back of the queue.

G. Contraction Heuristic Overview

This section introduces four different contraction planning heuristics in addition to the two cotengra contraction heuristics introduced in Section II-F.

NN-based	**Handcrafted**
Windowed-N	Lookahead
Tree Search	EMIT

Table 2: Overview of the different contraction planning heuristics introduced in Section III.

Table 2 shows which heuristics use neural network models and which do not. The principles behind each of them are outlined in the following:

- **Windowed-N**: Using the TDD size prediction model to make predictions and choose the smallest one in discrete windows.
- **Tree Search**: A weight function based on the TDD size prediction model is used for probabilities to make multiple possible plans.
- **Look-ahead**: Performing all available contractions and proceeding with the one that yields the smallest result.
- **EMIT**: Tensor network edges are added to a queue after each contraction to have contractions spread out throughout the tensor network.

IV. Results

To compare the efficiency of the proposed methods, several experiments are performed on well-known quantum circuits from a third-party benchmarking tool, MQT.BENCH [18]. The different methods are compared in terms of the time it takes to perform planning and contraction during quantum circuit equivalence checking on the selected circuits. Equivalence checking may fail if the space requirements of the contractions are exceeded, or if the equivalence result contradicts the ground truth. For all experiments, the space limit is naturally enforced by the computational power available. Failing by the wrong result is caused by implementation issues later discussed.

First, this section discusses the setup used for experimentation. Second, a comparison between the proof-of-concept Python implementation and the new C++ implementation is performed. Third, models are trained and evaluated, and then different models varying in the type of training data used are subsequently evaluated using the Windowed-N heuristic. Fourth, a comparison between the Windowed-N heuristic where the window size is varied is performed. Fifth, the tree search method is evaluated using different sample metrics. Sixth, the Lookahead heuristic is evaluated and compared to the Windowed-1 heuristic to see the effect of using actual sizes instead of predictions. Finally, the best heuristics, including the EMIT heuristic, are compared against the state-of-the-art equivalence checking tool Quantum Circuit Equivalence Checker from Munich Toolkit (QCEC) [3].

A. *Experimental Setup*

The implementation for the proposed methods is divided into a Python repository and a C++ repository. As shown in Figure 10, the C++ repository is mainly responsible for contractions and the planning heuristics that are used during contraction. Since some of the proposed contraction plan heuristics are implemented only in Python, the C++ repository is compiled into a library file, which is then loaded and run by Python. This allows Python-based planning heuristics to still utilise the faster contractions provided by the C++ repository. Contractions of TDDs in both Python and C++ are based on the implementations found at Veriqc/TDD and Veriqc/TDD_C, respectively. The implementations developed for this thesis can be found at Simonbolsen/P10 and ChBLA/TDDLinux for the Python and C++ implementations, respectively.

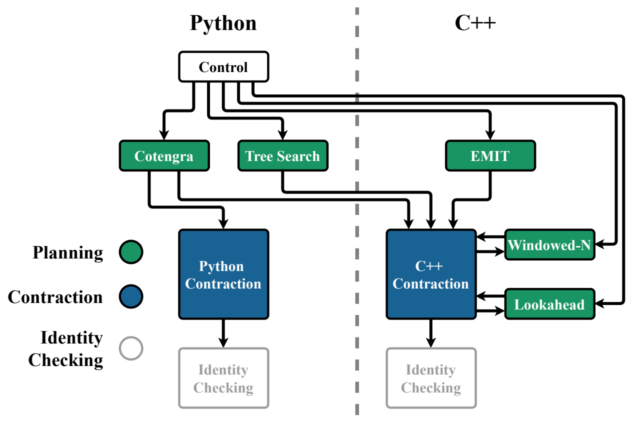

Fig. 10: Overview of the implementation and how the contraction planning heuristics are distributed between Python and C++. It is also shown how the Windowed and Lookahead heuristics are used during contraction.

To ensure comparable results with other research performed on the same topic of quantum circuit equivalence checking, the benchmark tool MQT.BENCH [18] is used. MQT.BENCH implements several established quantum circuits of varying difficulty and size. Additionally, the RandomEqv circuit is used, which is a random circuit where the number of gates is sampled from Gaussian distribution and gates are uniformly chosen from 12 different gates. Table 3 shows the circuits used as benchmarks. The provided circuits are scalable with regard to the number of qubits. This allows comparison of the same circuit but for an increasing number of qubits to estimate the complexities of the methods. MQT.BENCH also provides compilers such that the circuits can be expressed at four different abstraction levels. These abstraction levels are algorithmic, target-independent, target-dependent native gates, and target-dependent mapped, as ordered from most to least abstract.

For equivalence checking, two circuits are needed. Since these should be equivalent, the two circuits represent the same quantum algorithm but at different abstraction levels. We select the algorithmic layer and the target-dependent native gates layer for the evaluation.

Simple Circuits	Advanced Circuits
DJ	QFTEntangled
GHZ	RandomEqv
GraphState	RealAmpRandom
	WState

Table 3: The benchmark circuits used for the experiments from MQT.BENCH. They are split into Simple Circuits and Advanced Circuits based on a stark difference in the number of qubits it is possible to scale the circuits to.

For a baseline comparison of contraction planning, the state-of-the-art tensor network contraction planning tool Cotengra [8] is used. Particularly, the two contraction planning heuristics **Random-Greedy** and **Betweenness** supplied by Cotengra are used, as these seemingly are the best performing of the Cotengra methods for the setup used in this thesis. Additionally the heuristics discussed in Section III are also used.

All experiments are run using Python 3.9, CUDA 12.1, and C++17 on Ubuntu 20.04. The computer used has 20 GB available memory, an AMD Ryzen 5 1600 CPU, and an NVIDIA GeForce GTX 1060 6GB graphics card.

B. *Comparing Python and C++ Implementations*

As to increase the performance of the contraction part of the equivalence checking method, a C++ implementation is developed based on a pre-existing repository by Hong et. al [12]. The C++ implementation is evaluated and compared to the Python implementation to ascertain how much faster it is.

To properly evaluate the difference between Python and C++ regarding contraction, the same setup is used for both. The **Betweenness** heuristic is used for both Python and C++ as it is deterministic.

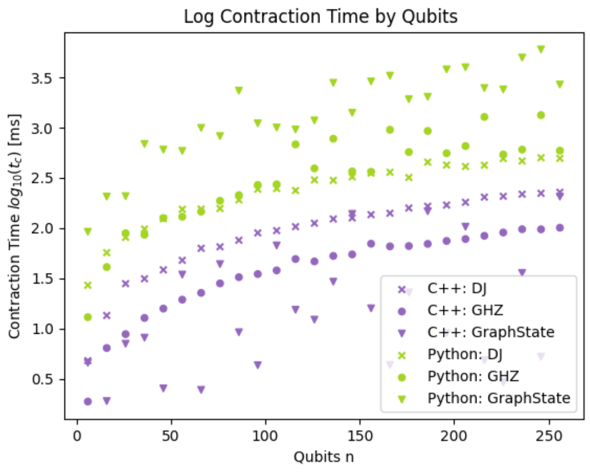

Fig. 11: Comparison of the total time spent on contracting all TDDs of the circuit in a log-scale between the Python and C++ implementations on DJ, GHZ, and GraphState. The evaluation starts at 6 qubits and goes up to 256 qubits with 10 qubit steps.

As seen in Figure 11, the C++ implementation is consistently faster at contracting than the Python implementation. For DJ the C++ implementation is 3-5 times faster, for GHZ C++ is consistently 10 times faster than the Python implementation,

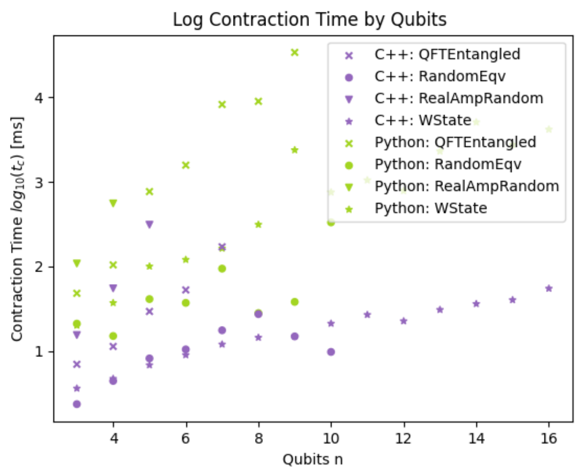

Fig. 12: Comparison of contraction time in a log-scale between the Python and C++ implementations on WState, QFTEntangled, RandomEqv, and RealAmpRandom. The evaluation starts at 3 qubits and goes up to 17 qubits for WState, 11 qubits for QFTEntangled and RandomEqv, and 7 qubits on RealAmpRandom.

and for GraphState the C++ implementation is 10-100 times faster than the Python implementation.

In Figure 12, the contraction times in a log-scale for WState, QFTEntangled, RandomEqv, and RealAmpRandom can be seen. While the C++ implementation is still typically faster, there is a lot more variance in the contraction times compared to the results in Figure 11. The C++ implementation appears to be between 3-100 times faster depending on the number of qubits and the circuit in question.

Finally, it can be seen from the results in Figures 11 and 12 that the C++ implementation provides a speed-up by a factor of 3-10 for most circuits and, for a select few, upwards of a 100 times speed-up. Consequently, the C++ implementation of TDD contractions is used for the remaining experiments.

C. Tensor Prediction Model

Training a model to predict the size of TDDs resulting from contraction requires training data with the actual sizes of the resulting TDDs. To obtain the training data and prevent overfitting on the benchmark circuits, random circuits are constructed and are contracted with the **Random-Greedy** Cotengra heuristic. Every contraction is then recorded and compiled into a dataset with the format used by the model, see Section III-A.

The random circuits the dataset is made from are constructed with 8 to 20 qubits and a few hundred additional circuits with up to 50 qubits. For each circuit, the amount of gates used is sampled from a Gaussian distribution. Then each gate is sampled from 12 different gates and placed on a uniformly chosen qubit. The gates are biased towards 2-qubit gates so that all qubits remain connected.

In total, there are $15,685,055$ contractions generated from the random circuits of which the dataset consists of the $2,660,957$ unique data points. The dataset is split into a 10% validation set and a 90% training set.

Hyper Parameters	Values
Depth	10
Embedding Size	64
Learning Rate	-2.9
Dropout Rate	$1.59 \cdot 10^{-3}$

Table 4: Hyperparameters for the TDD prediction models.

The variations of the tensor prediction models are trained with the Adam optimiser, the mean squared error loss function, a batch size of 2048, and an early stopping mechanism after 20 epochs without improvement. Table 4 shows the hyperparameters used for the models unless something else is stated. The hyperparameters were tuned on the validation data.

During training, learning rate decay is used. As seen in equation 32, the learning rate tends towards $10^{-2.9}$ as the number of elapsed epochs increases.

$$lr_{decay}(lr, \text{epoch}) = 10^{lr+a}$$
$$\text{where} \quad a = \frac{20}{25 + \text{epoch}^2} \tag{32}$$

Figure 13 shows how the amount of training data used impacts the validation loss. The model approaches a validation loss of 1. As loss is measured as mean squared error and all predicted TDD sizes are on a log_2 scale, the error on TDD size predictions is a factor of two. It is also seen that the variation in validation loss decreases dramatically between the three runs as the amount of data increases. This shows that the models trained on the full dataset are consistent in their training. The full dataset is also what is used hereafter unless explicitly stated otherwise.

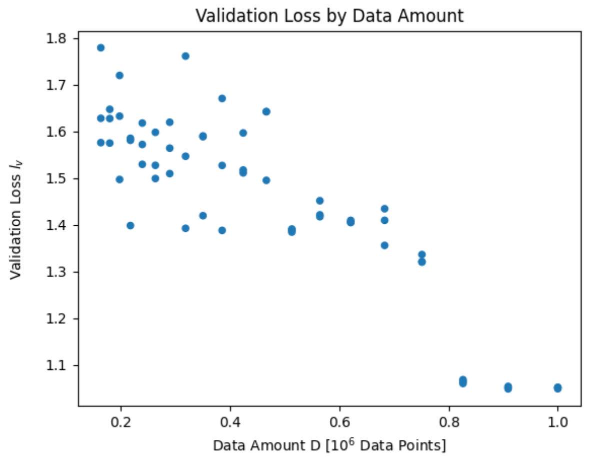

Fig. 13: Validation loss of the tensor prediction model by the amount of training data. Three runs are shown for each amount of data. The entire validation set is used for each run.

Figure 14 compares the predictions of the best-performing run from Figure 13 to their actual sizes. From this, it is clear that even if the circuits are constructed with up to 20 qubits, most contractions will be between smaller TDDs. Note that the maximum log size of a TDD is its number of indices.

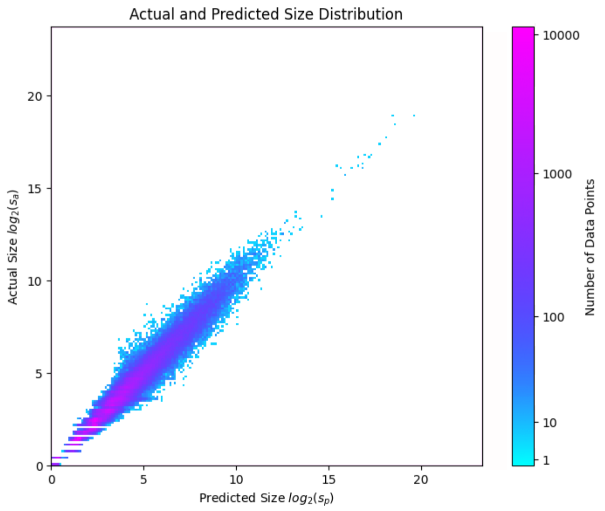

Fig. 14: The predicted sizes by the actual size on validation data using the best-performing model on the entire dataset.

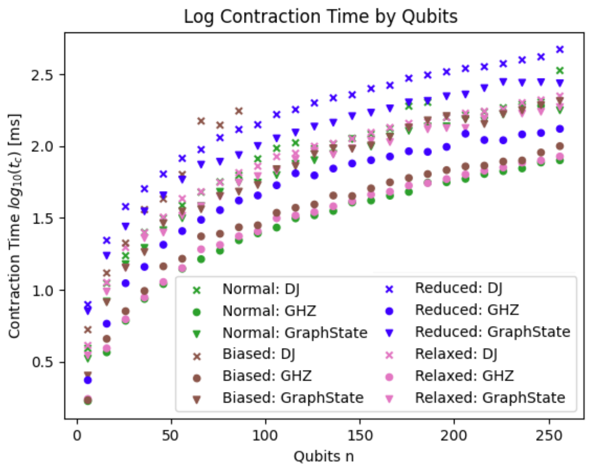

Fig. 15: Comparison of Greedy Prediction using *normal*, *biased*, *relaxed*, and *reduced* models on circuits GHZ, DJ, and GraphState.

D. Evaluation of Different Model Types

The Windowed-Max contraction planning heuristic initially introduced in Section III-B, must be evaluated using the models described in Section III-A and trained in Section IV-C. From Section IV-C it is shown that the accuracy of the predictions is on average off by a factor of two. Further, since the training and validation data consists of both contractions which results in larger and smaller TDDs than the operands, the final models are also trained to predict both. However, when paired with a greedy heuristic, accuracy in the prediction of large TDD sizes may become less valuable, as the smaller predictions are always prioritised.

The models evaluated here are different in the data that is used to train them. Three variations of the model are introduced here, where two of them has a bias towards being better at predicting contractions that result in smaller TDDs. While the overall amount of data decreases, the removed data may simply reduce the accuracy where it is needed. As such, four models are trained and evaluated with the Windowed-Max heuristic. One model uses all data for training referred to as *normal*. Another uses only contractions where the resulting size is less than the smallest of the two TDDs contracted referred to as *biased*. The third model uses all contractions where the size of the resulting TDD is less than or equal to twice the size of either of the two TDDs used for contraction referred to as *relaxed*. The fourth model, *reduced*, is trained on a uniform subset of the entire dataset with the same amount of data used by relaxed.

Figure 15 shows the performance of Windowed-Max using each of the three model types on the GHZ, DJ, and GraphState circuits. From the results, *reduced* is outperformed by the other models. Further, it appears that the *biased* model type is slightly outperformed by the other two which are almost equally good. The *reduced* model is trained on the same amount of data as *relaxed*, but *relaxed* performs significantly better. As the only difference between *normal* and *reduced* is the amount of data they are trained on, it indicates that it may be advantageous to bias the training data towards predictions of contractions towards contractions yielding smaller results. However, to better see the potential of the *relaxed* model type, significantly more training data of that type is required.

For the purposes of the remaining experiments, the *normal* model type is used as it performs slightly better than *relaxed*.

E. Tree Search

The experiments regarding tree search are presented in two steps. First, the hyperparameter of the weight function is tuned. Second, it is shown that the sample metrics from Definition 12 cannot be used to pick the best samples, and that tree search therefore is not viable in its current form.

Tree search makes several samples and chooses the contractions in these samples based on probability distribution from a weight function. The greedy weight function in Definition 11 has a parameter α which controls how much weight to assign to contractions based on the predicted size. The impact of α is shown in Figure 16 where it is demonstrated that the sample metrics and the amount of time spent on making several samples do not have an impact. The tree search heuristic is given some amount of time to produce samples and will finish the current sample when it runs out of time. In Figure 16 the runs that are given one second to plan produce a single sample and those that are given 60 seconds produce 9 samples to chose from. The two methods of choosing samples use the sum and maximum of the predictions on chosen contractions in the samples.

As the α value increases, the sampling with the greedy weight function comes closer to Windowed-Max and becomes more deterministic. The increased randomness in runs with lower α values increases the variation between the samples. Figure 16 shows that having several samples does not seem to decrease contraction time, no matter how much variation there is between the samples.

Because having several samples does not improve contraction time, we conclude that the sample metrics are unable to pick the correct samples. Figure 17 shows the two sample metrics,

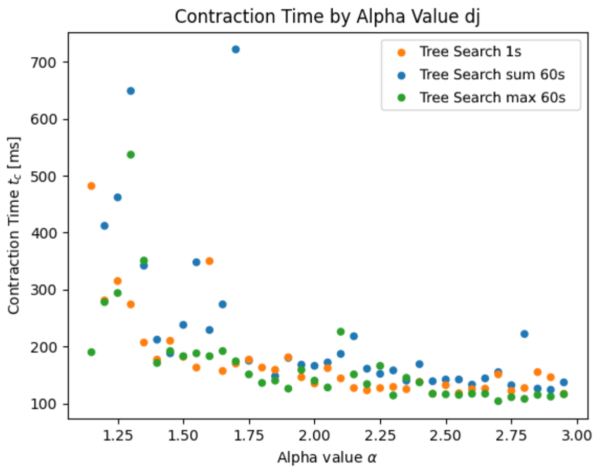

Fig. 16: The contraction time by the parameter α of the greedy weight function for three different versions of tree search. Evaluated on the DJ circuit with 128 qubits.

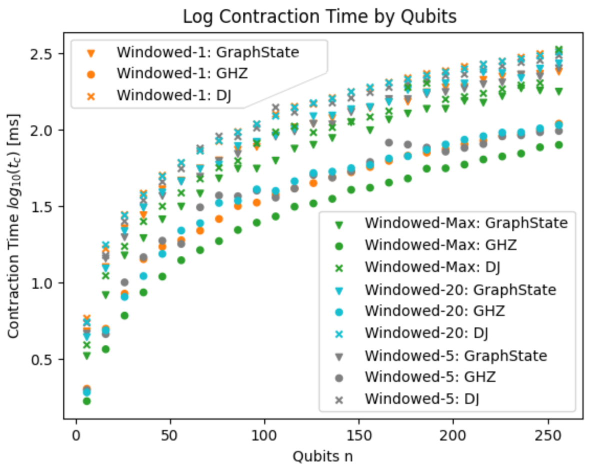

Fig. 18: Comparison on contraction time for the Windowed-N heuristic on circuits GHZ, DJ, and GraphState with window sizes of 1, 5, 20, and Max.

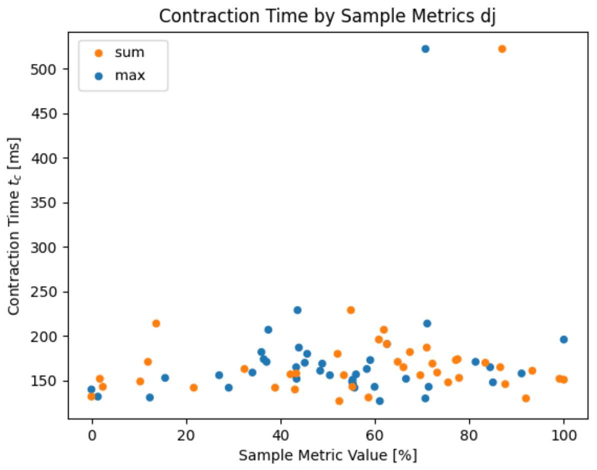

Fig. 17: The contraction time by the sample metrics. Both metrics are from the same contractions. As the metrics are on different scales they are both visualised with their smallest value at 0% and their largest value at 100%.

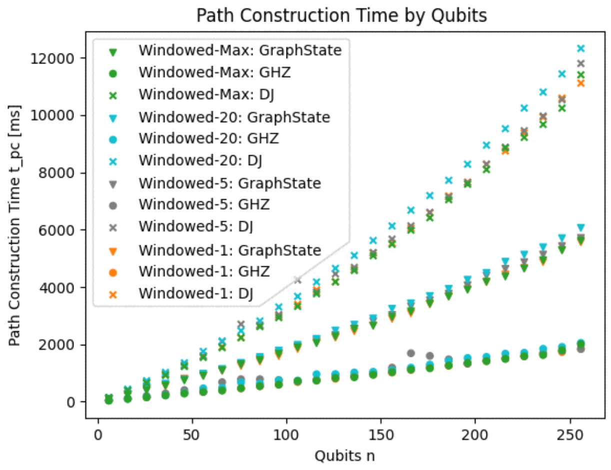

Fig. 19: Comparison on planning time for the Windowed-N heuristic on circuits GHZ, DJ, and GraphState with window sizes of 1, 5, 20, and Max.

sum and max for 40 repeats of the same setup, with a single sample each, $\alpha = 2.0$, and DJ with 128 qubits. It can be seen that the sample metrics cannot be used to choose between samples as they are independent of the contraction time.

Therefore, we find that the tree search heuristic for the current model has no advantage over the other greedy heuristics.

F. *Windowed-N Heuristics*

As the Windowed-N heuristic varies how often the graph is updated with actual values, it is relevant to evaluate what impact it has on contraction speed. As such, the effect that window size has on the time spent both contracting and planning is evaluated. Here, the window sizes 1, 5, 20, and Max are used, where Max is the length of the entire contraction plan.

Figure 18 shows the Windowed-N heuristic on the GHZ, DJ, and GraphState circuits with different window sizes. Intuition suggests that the contraction time should increase with the window size since more predictions rely on possibly inaccurate previous predictions. However, as seen in Figure 18, there is no significant difference in contraction time for the three window sizes 1, 5, and 20. Additionally, it appears, counter-intuitively, that the Windowed-Max, which relies on most previous predictions, performs the best with regards to contraction time. As seen in Figure 19 there also appears to be no significant difference in the planning time, ultimately meaning that, for the window sizes used, the window sizes of 1, 5, and 20 perform similarly, and that the Windowed-Max performs the best.

G. *Lookahead*

The alternative to using neural networks to predict the sizes of the TDDs resulting from contraction is to compute the TDDs and measure their sizes. This heuristic is referred to

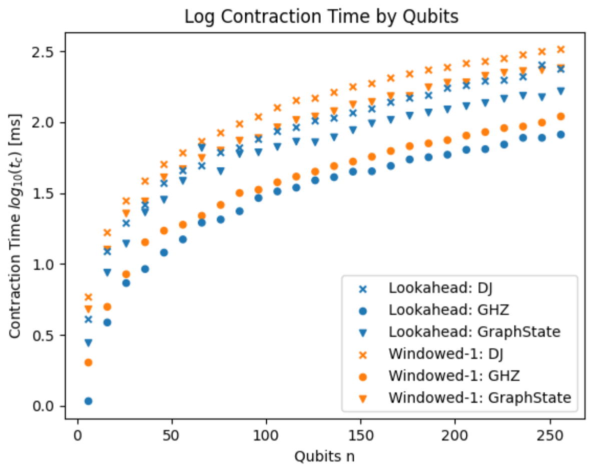

Fig. 20: Comparison of contraction time for Windowed-1 and Lookahead on circuits DJ, GHZ, and GraphState on a log scale.

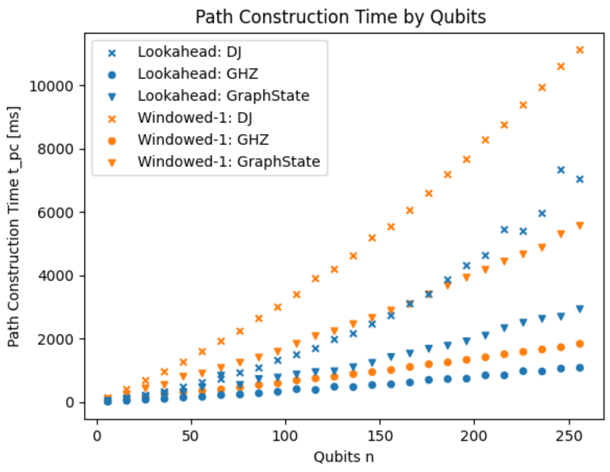

Fig. 21: Comparison of planning time for Windowed-1 and Lookahead on circuits DJ, GHZ, and GraphState.

as *Lookahead* in Section III-E, as it computes all currently available contractions and then greedily chooses the smallest. The main concern for such a heuristic is the potentially exponential cost of computing some contraction that could otherwise have been avoided. The immediate benefits of such a heuristic are that the actual sizes of contractions are known and contractions on many TDDs can be performed fast, as observed during previous experiments.

In Figure 20 the results of Lookahead and Windowed-1 are shown with regards to the total contraction time. Windowed-1 is selected for comparison as it is the heuristic most similar to Lookahead. The difference between Lookahed and Windowed-1 is that Lookahead contracts the TDDs and measures them, whereas Windowed-1 uses the TDD size prediction model. As can be seen, the Lookahead heuristic consistently outperforms the Windowed-1 heuristic, with roughly the same speed-up achievable on each of the shown circuits.

Since Lookahead may suffer from exponentially difficult contractions during planning, Figure 21 shows the time spent

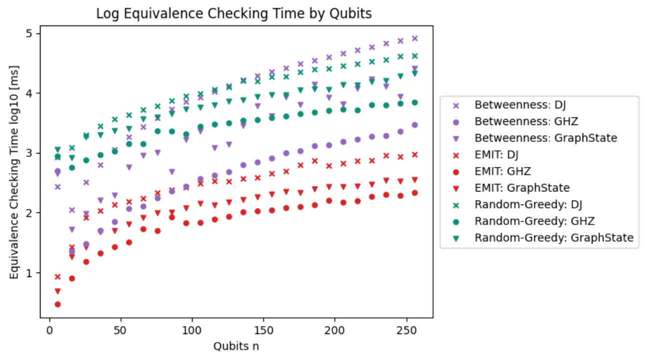

Fig. 22: Comparison in sum of planning and contraction time for EMIT, Cotengra Betweenness, and Cotengra Random-Greedy on circuits GHZ, DJ, and GraphState.

planning for the same runs as Figure 20. Again, Lookahead outperforms Windowed-1, especially so for larger circuits. The potentially exponential contractions do not appear to impede the Lookahead heuristic during planning. Similar results are achieved for circuits WState, QFTEntangled, and RealAmpRandom.

H. Comparison with State-Of-The-Art

Given the increased performance both from using a faster implementation of contraction and also improving contraction paths, it is instructive to compare the best performing runs from the previous work with the now best performing methods. Additionally, QCEC is included in the comparison, as it represents the current state-of-the-art. However, to make the results comparable, QCEC is configured such that it does not perform optimisations and reductions using methods such as ZX-calculus. It therefore only relies on its QDD-based checker. Notably, QCEC does not itself report data on the time resources of the equivalence checking, and there is no measurable way to determine how much time is spent planning, contracting, or on miscellaneous tasks. As such, the entire time QCEC spends equivalence checking is recorded. The methods developed for this thesis consider only time spent planning and contracting, as these are the measurements that are being optimised. As such, a direct comparison between QCEC and the other methods may underestimate the performance of QCEC.

Figure 22 shows the performance on the combined time spent planning and contracting on circuits GHZ, DJ, and GraphState between EMIT and the two cotengra heuristics, **Betweenness** and **Random-Greedy**. EMIT performs significantly better than both cotengra heuristics, often 10 times better and for DJ with many qubits more than 100 times better. Similar results are achieved on the advanced circuits, which clearly shows that EMIT is significantly better than the cotengra heuristics.

Figure 23 shows the performance of equivalence checking for Lookahead, EMIT, and QCEC on DJ, GHZ, and GraphState circuits. The QCEC outperforms Lookahead and EMIT on DJ and GHZ, but fails to surpass 116 qubits on GraphState, where both Lookahead and EMIT manage 256 qubits with no problems. For GHZ it appears that EMIT performs better than QCEC on circuits with more qubits, but since the

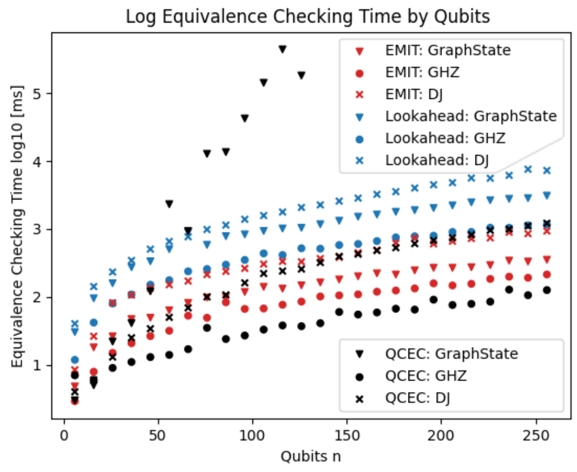

Fig. 23: Comparison of the best performing heuristic against QCEC in regards to the sum of contraction time and planning time on circuits GHZ, DJ, and GraphState.

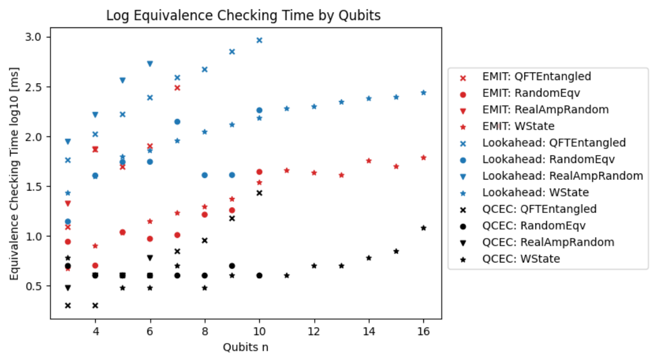

Fig. 24: Comparison of the best performing heuristic against QCEC in regards to the sum of contraction time and planning time on circuits WState, QFTEntangled, RandomEqv, and RealAmpRandom.

time measurement for QCEC potentially includes several miscellaneous tasks, it is likely not entirely accurate. However, EMIT does appear to scale better than QCEC, so the difference may become more noticeable on much larger circuits. The results show that EMIT is about 10 times faster than Lookahead, which is mainly attributed to faster planning time.

Figure 24 shows the performance of equivalence checking for Lookahead, EMIT, and QCEC on WState, QFTEntangled, RandomEqv, and RealAmpRandome circuits. Compared to Figure 23, the results are more inconsistent; however, it still appears that QCEC is the best followed by EMIT. Lookahead is worse than either except on QFTEntangled where it beats EMIT by scaling to a higher number of qubits.

Since the methods appear to scale well on DJ, GHZ, and GraphState, the limitations of the number of qubits are also evaluated. It appears that the developed method, no matter the heuristic, cannot exceed 667 qubits on GHZ, 266 qubits on DJ, and 416 qubits on GraphState due to memory issues when creating tensor networks. QCEC manages to exceed 2500 qubits for both GHZ and DJ, but as also seen earlier, does not exceed 116 qubits on GraphState.

V. Discussion

Several aspects pertaining to the results of the previous Section IV are worth further discussion. These particularly involve the training and performance of the neural network models and the performance of the contractions. As these aspects affect the overall performance of the evaluated methods, addressing these may result in increased performance or a more stable method.

A. *Float Precision in TDD Contractions*

The current C++ implementation used for contraction suffers from floating point inaccuracy during contraction of TDDs. This sometimes leads to incorrect results regarding the final TDD, as some of the weights are not perfectly 0 or 1, but rather slightly different. While this constitutes an implementation deficiency and is not directly related to the methodology, it does affect the results. Particularly, the circuits that use random parameters for gates and many gates per qubit are affected by this, as the inaccuracies propagate when contracting.

While the implementation allows the precision of floating point operations to be specified regarding TDD contraction, no one amount of precision works best for all circuits. As such, different precision is required for different circuits, and often one must try several to get the correct precision where the correct result is yielded. For the purposes of evaluating the heuristics, it is not detrimental, as it is known that the circuits are equivalent and any result not agreeing with such can be discarded. It does, however, mean that for some methods, there are circuits that cannot be evaluated, as they consistently fail to be proven equivalent due to no precision fitting those circuits.

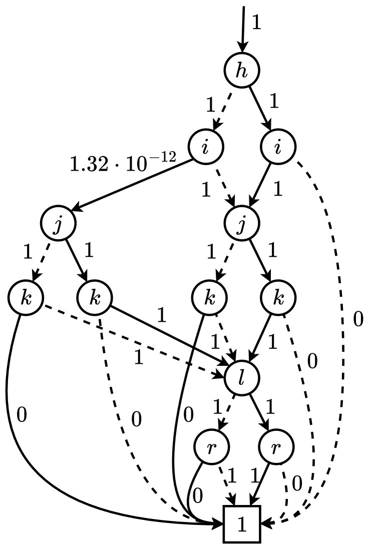

Fig. 25: A TDD depicting the potential effects of imprecision regarding floating point operations when contracting circuits. The TDD would be identity if the $1.32 \cdot 10^{-12}$ weight was correctly evaluated to 0.

Figure 25 shows a typical result (scaled down) of floating point issues with regards to the resulting TDD of contraction of two equivalent circuits. As can be seen, if the edge denoted with a weight of $1.32 \cdot 10^{-12}$ was instead 0, the TDD depicted would be identity.

Finally, the degree to which floating point issues affect the contractions of a circuit appears to be connected to the path, as some heuristics are more affected than others. It is not known why some paths are more prone to floating point issues, but finding and fixing the root cause for this may significantly improve the stability of the contraction implementation.

B. Issues Regarding Data Generation

To train the neural network models, a significant amount of data is needed. As the data needed must be contracted between two TDDs with information on the resulting TDD for evaluation purposes, actual contractions must be performed. In order for the model to be able to predict accurately, it must be trained on a large sample of TDD contractions. TDD contractions are not only dependent on the circuit used but also on the contraction plan used on said circuit. Given the strict need for performing actual contractions in order to generate proper data for training, there is a limit to the scale of the TDDs that can be generated data for. Since TDDs scale exponentially in the worst case, there is a computational limit with regards to the resources for how large the resulting TDDs can be.

To further complicate the generation of proper training data, both circuits and heuristics must be chosen, such that there is as little overfitting for the circuits and heuristics used during experimentation as possible. While the heuristic has some bearing on the data generation, it is ultimately judged to not be as important to keep different as the circuit, since enough variation of the circuit should remove possible overfitting from the heuristic. Since using any of the circuits also used for the experiments may bias the models towards performing unjustly well on some circuits, random equivalent circuits are used for data generation. Though using random equivalent circuits should remove any bias towards some circuits, it does mean that the qubit limitation for circuits that can be contracted is reduced, as the random equivalent circuits are difficult to perform equivalence checking for.

The main issue with regards to not being able to increase the number of qubits for the circuits for which data generation is based on, is that the TDD structures are often very different between the easier circuits with more qubits and the harder circuits for which fewer qubits are possible. For easier circuits with more qubits, the TDDs are typically tall but slim and with many indices; both shared and not. The harder circuits with fewer qubits are typically the opposite, where the TDDs are wider in structure with fewer both shared and non-shared indices. This effect is illustrated in Figure 26, where the structure of the two portrayed TDDs are very different. Note that the width of a TDD can be exponential in the height in the worst case.

C. Neural Network Models

For the heuristics that rely on predictions to generate a valid path, we trained and evaluated several neural network models using different architectures and different amounts of data. While several models are trained with varying structure, depth, and data amounts, there are still unexplored models that may perform better than the models used for experimentation.

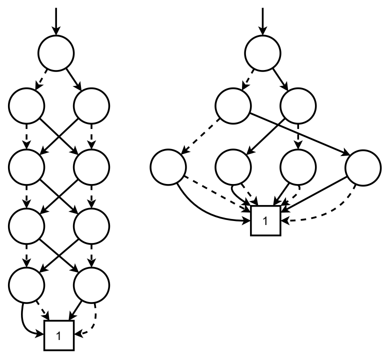

Fig. 26: Depiction of a tall but slim TDD and a small but wide TDD, respectively. The taller TDDs are more likely to occur during contraction of circuits with more qubits, while the wider TDDs are more likely to occur during contraction on more difficult circuits which have fewer qubits.

The models used for experimentation are based on one configuration of inputs extracted from the TDDs which are to be contracted. Specifically, the gates, indices, and sizes of the two TDDs are used along with the number of indices they share. While this input format does prove to work, as seen from the experiments, it fails to include information such as gate parameters, TDD structure, and where in the TDDs the shared indices are located. All this information may improve the predictive power of the models if sufficient data is generated.

1) Parameterised Gates: Some circuits contain parameterised gates and the neural network model does not consider these parameters when making predictions. Since the parameters are real values and vastly change the effect of gates depending on the actual value, the size of the resulting contraction may vary greatly depending on the parameters used. Consider for example the quantum gate $R_x(\theta)$ which specifies rotation around the x-axis. For two different parameters, the transformation of the gate completely changes, as seen by:

$$R_x(0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad R_x(\pi) = \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix} \tag{33}$$

In the data on which the model is trained only a single value is used for each parameter, meaning that all parameterised gates of the same type are equivalent in the dataset. Some generality is therefore lost compared to actual quantum circuits.

2) Expanding the Model: The main way to improve the model is to give it more information about the TDDs and the gates they are made up of. This information includes both the position of shared indices in the TDDs and the parameters of parameterised gates. The difficulty of expanding the model to accommodate these is that they both consist of a variable amount of information. In contrast to the current input vectors of constant size, to use the additional data the model would have to be able to take any number of shared indices positions and any number of gate parameters.

3) Graph Neural Network: Initially, a graph neural network was designed and trained to emulate a planning heuristic by taking the structure of the tensor networks as input. This was done with the eventual goal of using reinforcement learning to go beyond existing heuristics. This approach was not pursued further as it was found that training and prediction were far too slow to be viable.

Using graph neural networks on the graph structure of TDDs may also make it possible to make more accurate size predictions. This would likely also be considerably slower than the current prediction model and it would not be possible to use it for predicting further than one step ahead of contraction.

VI. Conclusion

This thesis looked at a quantum circuit equivalence checking method using tensor network contraction plans with Tensor Decision Diagrams (TDDs). The thesis was built upon a previous work of the same authors and further investigates the effects of contraction plans for equivalence checking using TDDs. Where the previous work used simple and pre-existing heuristics to provide a proof-of-concept of the equivalence checking method, this work focused on developing new contraction planning heuristics. Previously regarded contraction planning heuristics were not tailored to work well with TDDs, as there were no considerations toward the sizes of the intermediate TDDs during contraction. More practically, the previous experimentation was done with a lacking implementation which greatly decreased the performance of the then proposed methodology.

In this thesis, we addressed the reservations about the previous paper. A new implementation was used, showing between 3 and 100 times speed-up of TDD contractions. Neural network models were designed and trained to predict the size of a TDD resulting from a contraction. The trained models often predicted within a factor of two of the correct sizes but were seemingly not complex enough to predict when TDDs become smaller by contraction. Despite the neural network models lacking in predictive accuracy, the Windowed heuristics, which utilises a prediction model to make greedy choices, was comparable to state-of-the-art contraction planning heuristics when doing equivalence checking using TDDs. Compared to the existing cotengra heuristics it is found that our model-based and handcrafted heuristics are faster. The EMIT heuristic combined with the new C++ implementation has vastly improved the equivalence checking method of TDD contraction and made it competitive with the state-of-the-art equivalence checking tool QCEC.

A. *Future Work*

While the neural network heuristics do perform similarly to the best cotengra heuristics for equivalence checking using TDDs, there is still much potential to further improve on both the models and the heuristics. Currently, the neural network models appear to be incapable of accurately predicting TDDs where the operands are both larger.

One way to potentially increase the predictive power of the neural network model is to include more information regarding contractions. Adding additional information such as the positions of the shared indices and the parameters of parameterised gates may provide the necessary information for improving accuracy.

Another aspect of the thesis that may be worth investigating further is the current floating point imprecision, which, for some circuits, results in wrongful results. One approach may be to extend the TDD formalism with intervals as opposed to the current floating point values. Such intervals may allow some uncertainty to be represented both from noisy quantum gates and floating point imprecision in any implementation.

The queue heuristic also has unexplored potential in how tensor network edges are added to the queue. Currently, these edges are added indiscriminately and their order comes down to implementation. Alternatively, mechanisms could be used to favour TDDs with fewer indices.

Acknowledgment

The authors would like to thank supervisor Christian Schilling and co-supervisor Kim Guldstrand Larsen for help and guidance throughout the thesis.

References

[1] Afshin Abdollahi and Massoud Pedram. Analysis and synthesis of quantum circuits by using quantum decision diagrams. In Georges G. E. Gielen, editor, *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, pages 317–322. European Design and Automation Association, Leuven, Belgium, 2006.

[2] Lukas Burgholzer, Alexander Ploier, and Robert Wille. 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.*, 42(4):1113–1122, 2023.

[3] Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 40(9):1810–1824, 2021.

[4] Cristian S. Calude and Elena Calude. The road to quantum computational supremacy. In David H. Bailey, Naomi Simone Borwein, Richard P. Brent, Regina S. Burachik, Judy-anne Heather Osborn, Brailey Sims, and Qiji J. Zhu, editors, *From Analysis to Visualization*, pages 349–367, Cham, 2020. Springer International Publishing.

[5] Bob Coecke and Ross Duncan. Interacting quantum observables. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2008.

[6] Olivia Di Matteo, Santiago Núñez-Corrales, Michał Stechły, Steven P. Reinhardt, and Tim Mattson. An Abstraction Hierarchy Toward Productive Quantum Programming. *arXiv e-prints*, page arXiv:2405.13918, May 2024.

[7] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, 1939.

[8] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, mar 2021.

[9] Xin Hong, Wei-Jia Huang, Wei-Chen Chien, Yuan Feng, Min-Hsiu Hsieh, Sanjiang Li, Chia-Shun Yeh, and Mingsheng Ying. Decision diagrams for symbolic verification of quantum circuits. In Brian La Cour, Lia Yeh, and Marek Osinski, editors, *IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023*, pages 970–977. IEEE, 2023.

[10] Xin Hong, Wei-Jia Huang, Wei-Chen Chien, Yuan Feng, Min-Hsiu Hsieh, Sanjiang Li, and Mingsheng Ying. Equivalence checking of parameterised quantum circuits, 2024.

[11] Xin Hong, Mingsheng Ying, Yuan Feng, Xiangzhen Zhou, and Sanjiang Li. Approximate equivalence checking of noisy quantum circuits. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 637–642. IEEE, 2021.

[12] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. A tensor network based decision diagram for representation of quantum circuits. *ACM Trans. Design Autom. Electr. Syst.*, 27(6):60:1–60:30, 2022.

[13] DOMINIK JANZING, PAWEL WOCJAN, and THOMAS BETH. ”non-identity-check” is qma-complete. *International Journal of Quantum Information*, 03(03):463–473, 2005.

[14] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Process. Lett.*, 7(2):157–168, 1997.

[15] Christian Bøgh Larsen and Simon Brun Olsen. Quantum equivalence checking using tensor decision diagrams and contraction paths. 2023.

[16] Eli A. Meirom, Haggai Maron, Shie Mannor, and Gal Chechik. Optimizing tensor network contraction using reinforcement learning. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 15278–15292. PMLR, 2022.

[17] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking of quantum circuits with the ZX-calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 12(3):662–675, sep 2022.

[18] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT bench: Benchmarking software and design automation tools for quantum computing. *CoRR*, abs/2204.13719, 2022.

[19] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking software and design automation tools for quantum computing. *Quantum*, 2023. MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[20] Nils Quetschlich, Florian J. Kiwit, Maximilian A. Wolf, Carlos A. Riofrio, Lukas Burgholzer, Andre Luckow, and Robert Wille. Towards application-aware quantum circuit compilation, 2024.

[21] Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for deep quantum circuits on NISQ processors with 100+ qubits. *CoRR*, abs/2403.11598, 2024.

[22] George F. Viamontes, Igor L. Markov, and John P. Hayes. Checking equivalence of quantum circuits and states. In Georges G. E. Gielen, editor, *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, pages 69–74. IEEE Computer Society, 2007.

[23] Thorsten B. Wahl and Sergii Strelchuk. Simulating quantum circuits using efficient tensor network contraction algorithms with subexponential upper bound. *Phys. Rev. Lett.*, 131:180601, Oct 2023.

[24] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. *Decision Diagrams for Quantum Computing*, pages 1–23. Springer International Publishing, Cham, 2023.

[25] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.