Data structures for quantum circuit verification and how to compare them

Vinkhuijzen, L.T.

Chapter 3

LIMDD: a decision diagram for simulation of quantum computing including stabilizer states

Efficient methods for the representation and simulation of quantum states and quantum operations are crucial for the optimization of quantum circuits. Decision diagrams (DDs), a well-studied data structure originally used to represent Boolean functions, have proven capable of capturing relevant aspects of quantum systems, but their limits are not well understood. In this work, we investigate and bridge the gap between existing DD-based structures and the stabilizer formalism, an important tool for simulating quantum circuits in the tractable regime. We first show that although DDs were suggested to succinctly represent important quantum states, they actually require exponential space for certain stabilizer states. To remedy this, we introduce a more powerful decision diagram variant, called Local Invertible Map-DD (LIMDD). We prove that the set of quantum states represented by poly-sized LIMDDs strictly contains the union of stabilizer states and other decision diagram variants. Finally, there exist circuits which LIMDDs can efficiently simulate, while their output states cannot be succinctly represented by two state-of-the-art simulation paradigms: the stabilizer

decomposition techniques for Clifford + T circuits and Matrix-Product States.

This chapter contributes to answering Research Question 1:

> **Research question 1.** *Can we unite the strengths of decision diagrams and the stabilizer formalism?*

This chapter answers this question by uniting the strengths of decision diagrams and the stabilizer formalism in a new data structure, the LIMDD. This new DD can efficiently simulate any circuit that can also be efficiently simulated by an existing state-of-the-art DD, the QMDD, and moreover can simulate all stabilizer circuits in polynomial time and represent all stabilizer states in polynomial space. By uniting two successful approaches, LIMDDs thus pave the way for fundamentally more powerful solutions for simulation and analysis of quantum computing.

3.1 Introduction

Classical simulation of quantum computing is useful for circuit design [74,373], verification [71,75] and studying noise resilience in the era of Noisy Intermediate-Scale Quantum (NISQ) computers [267]. Moreover, identifying classes of quantum circuits that are classically simulatable, helps in excluding regions where a quantum computational advantage cannot be obtained. For example, circuits containing only Clifford gates (a non-universal quantum gate set), using an all-zero initial state, only compute the so-called 'stabilizer states' and can be simulated in polynomial time [3,109,135,136,325]. Stabilizer states, and associated formalisms for expressing them, are fundamental to many quantum error correcting codes [135] and play a role in measurement-based quantum computation [271]. In fact, simulation of universal quantum circuits is fixed-parameter tractable in the number of non-Clifford gates [64], which is why many modern simulators are based on *stabilizer decomposition* [61,62,64,165,184,185].

Another method for simulating universal quantum computation is based on decision diagrams (DDs) [7,67,69,334], including Algebraic DDs [27,84,124,331], Affine Algebraic DDs [281], Quantum Multi-valued DDs [227,374], and Tensor DDs [163]. A DD is a directed acyclic graph (DAG) in which each path represents a quantum amplitude, enabling the succinct (and exact) representation of many quantum states through the combinatorial nature of this structure. A DD can also be thought of as a homo-

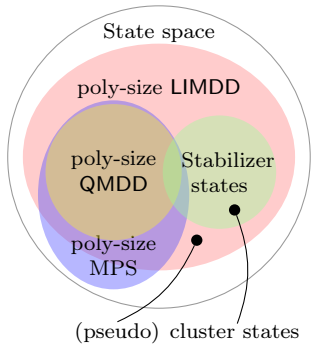


Figure 3.1: The set of stabilizer states and states representable as poly-sized: (Pauli-)LIMDDs (this work), QMDDs and MPS.

morphic (lossless) compression scheme, since various *manipulation operations for DDs* exist which implement any quantum gate operation, including measurement (without requiring decompression). Strong simulation is therefore easily implemented using a DD data structure [163,227,374]. Indeed, DD-based simulation was empirically shown to be competitive with state-of-the-art simulators [157,334,374] and is used in several simulator and circuit verification implementations [162, 332]. DDs and the stabilizer formalism are introduced in Section 3.2.

QMDDs are currently the most succinct DD supporting quantum simulation, but in this chapter we show that they require exponential size to represent a type of stabilizer state called a cluster state [65]. In order to unite the strengths of DDs and the stabilizer formalism and inspired by SLOCC (Stochastic Local Operations and Classical Communication) equivalence of quantum states [82, 107], in Section 3.3, we propose LIMDD: a new DD for quantum computing simulation using local invertible maps (LIMs). Specifically, LIMDDs eliminate the need to store multiple states which are equivalent up to LIMs, allowing more succinct DD representations. For the local operations in the LIMs, we choose Pauli operations, creating a Pauli-LIMDD, which we will simply refer to as LIMDD. We prove that there is a family of quantum states —called pseudo cluster states— that can be represented by poly-sized (Pauli-)LIMDDs but that require exponentially-sized QMDDs and cannot be expressed in the stabilizer formalism. We also show the same separation for matrix product states (MPS) [262,335,346]. Figure 3.1 visualizes the resulting separations.

Further, we give algorithms for simulating quantum circuits using Pauli-LIMDDs. We continue by investigating the power of these algorithms compared to state-of-the-

art simulation algorithms based on QMDD, MPS and stabilizer decomposition. We find circuit families which Pauli-LIMDD can efficiently simulate, which stands in stark contrast to the exponential space needed by QMDD-based, MPS-based and a stabilizer-decomposition-based simulator (the latter result is conditional on the exponential time hypothesis). This is the first analytical comparison between decision diagrams and matrix product states.

Efficient decision diagram operations for both classical [97] and quantum [74] applications crucially rely on *dynamic programming* (storing the result of each intermediate computation) and *canonicity* (each quantum state has a unique, smallest representative as a LIMDD) [59,183,303]. We provide algorithms for both in Section 3.4. Indeed, the main technical contribution of this chapter is the formulation of a canonical form for Pauli-LIMDDs together with an algorithm which brings a Pauli-LIMDD into this canonical form. By interleaving this algorithm with the circuit simulation algorithms, we ensure that the algorithms act on LIMDDs that are canonical and as small as possible.

The canonicity algorithm effectively determines whether two n-qubit quantum states $|\varphi\rangle, |\psi\rangle$, each represented by a LIMDD node φ, ψ, are equivalent up to a Pauli operator P, i.e, $|\varphi\rangle = P|\psi\rangle$, which we call an isomorphism between $|\varphi\rangle$ and $|\psi\rangle$. Here $P = P_n \otimes ... \otimes P_1$ consists of single qubit Pauli operators P_i (ignoring scalars for now). In general, there are multiple choices for P, so the goal is to make a deterministic selection among them, to ensure canonicity of the resulting LIMDD. To do so, we first take out one qubit and write the states as, e.g., $|\varphi\rangle = c_0 |0\rangle |\varphi_0\rangle + c_1 |1\rangle |\varphi_1\rangle$ for complex coefficients c_0, c_1. We then realize that $P_{\text{rest}} = P_{n-1} ... \otimes P_1$ must map the pair $(|\varphi_0\rangle, |\varphi_1\rangle)$ to either $(|\psi_0\rangle, |\psi_1\rangle)$ or $(|\psi_1\rangle, |\psi_0\rangle)$ (in case P_n is a diagonal or antidiagonal, respectively). Hence P_{rest} is a member of the intersection of the two sets of isomorphisms. Next, we realize that the set of all isomorphisms, e.g. mapping $|\varphi_0\rangle$ to $|\psi_0\rangle$, is a coset $\pi \cdot G$ of the stabilizer group G of $|\varphi_0\rangle$ (i.e. the set of isomorphisms mapping $|\varphi_0\rangle$ to itself) where π is a single isomorphism $|\varphi_0\rangle \to |\psi_0\rangle$. Thus, to find a (canonical) isomorphism between n-qubit states $|\varphi\rangle \to |\psi\rangle$ (or determine no such isomorphism exists), we need three algorithms: to find (a) an isomorphism between $(n-1)$-qubit states, (b) the stabilizer group of an $(n-1)$-qubit state (in fact: the group generators, which form an efficient description), (c) the intersection of two cosets in the Pauli group (solving the *Pauli coset intersection problem*). Task (a) and (b) are naturally formulated as recursive algorithms on the number of qubits, which invoke each other in the recursion step. For (c) we provide a separate algorithm which first

rotates the two cosets such that one is a member of the Pauli Z group, hence isomorphic to a binary vector space, followed by using existing algorithms for binary coset (hyperplane) intersection. Having characterized all isomorphisms $|\varphi\rangle \to |\psi\rangle$, we select the lexicographical minimum to ensure canonicity. We emphasize that the algorithm works for arbitrary quantum states, not only stabilizer states.

In Chapter 4, we describe a software implementation of Pauli-LIMDDs. We evaluate this implementation empirically against QMDDs in a case study in which we simulate a quantum circuit which applies a quantum Fourier Transform to a random stabilizer state.

To further establish a separation between LIMDDs and stabilizer rank-based simulators, we provide a small numerical experiment which tries to find a low-rank stabilizer decomposition for the so-called Dicke state, in Section 3.6. A state-of-the-art algorithm fails to find a low-rank decomposition, whereas LIMDDs represent and prepare such states efficiently; a proof is given in App. D.

3.2 Preliminaries: decision diagrams and the stabilizer formalism

Here, we briefly introduce two methods to manipulate and succinctly represent quantum states: decision diagrams, which support universal quantum computing, and the stabilizer formalism, in which a subset of all quantum computations is supported which can however be efficiently classically simulated. Both support *strong simulation*, i.e. the probability distribution of measurement outcomes can be computed (through *weak simulation* one only samples measurement outcomes).

For an introduction to quantum computing, see Section 2.2; for an introduction to decision diagrams, see Section 2.3

3.2.1 Decision diagrams

An n-qubit quantum state $|\varphi\rangle$ can be represented as a 2^n-dimensional vector of complex numbers (modeling amplitudes) and can thus be described by a pseudo-Boolean

function $f : \{0,1\}^n \to \mathbb{C}$ where

$$|\varphi\rangle = \sum_{x_1,\ldots,x_n \in \{0,1\}} f(x_n, ..., x_1) \, |x_n\rangle \otimes ... \otimes |x_1\rangle \, . \tag{3.1}$$

The Quantum Multi-valued Decision Diagram (QMDD) [227] is a data structure which can succinctly represent functions of the form $f\colon \{0,1\}^n \to \mathbb{C}$, and thus can represent any quantum state per Equation 3.1. A QMDD is a rooted DAG with a unique leaf node $\boxed{1}$, representing the value 1. Figure 3.2 (d) shows an example (and its construction from a binary tree). Each node has two outgoing edges, called its *low edge* (dashed line) and its *high edge* (solid line). The diagram has *levels* as each node is labeled with (the index of) a variable; the root has index n, its children $n-1$, etc, until the leaf with index 0 (the set of nodes with index k form level k). Hence each path from root to leaf visits nodes representing the variables x_3, x_2, x_1 in order. The value $f(x_n, \ldots, x_1) = \langle x_n ... x_1 | \varphi \rangle$ is computed by traversing the diagram, starting at the root edge and then for each node at level i following the low edge (dashed line) when $x_i = 0$, and the high edge (solid line) when $x_i = 1$, while multiplying the edge weights (shown in boxes) along the path, e.g., $f(1,1,0) = \frac{1}{2} \cdot 1 \cdot -\sqrt{2} \cdot 1 = -\frac{1}{\sqrt{2}}$ in Figure 3.2.

A path from the root to a node v with index k (on level k) thus corresponds to a *partial assignment* $(x_n = a_n, \ldots, x_{k-1} = a_{k-1})$, which induces subfunction $f_{\vec{a}}(x_k, \ldots, x_1) \triangleq f(a_n, \ldots, a_{k-1}, x_k, \ldots, x_1)$. The node v represents this subfunction *up to a complex factor* γ, which is stored on the edge incoming to v along that path. This allows any two nodes which represent functions equal up to a complex factor to be *merged*. For instance, the node u on level 1 in Figure 3.2 represents $f_{01} = f_{10} = \frac{-1}{\sqrt{2}} f_{11} = 0 \cdot f_{00}$. When all eligible nodes have been merged, the QMDD is *reduced*. A reduced QMDD is a *canonical* representation: a given function has a unique reduced QMDD.

Canonicity ensures that the QMDD is always as small as possible as redundant nodes are merged. But more importantly, canonicity allows for quick equality checks: two diagrams represent the same state *if and only if* their root edges are the same (i.e., have the same label and point to the same root node). This allows for efficient QMDD manipulation algorithms (i.e. updating the QMDD upon performing a gate or measurement) through dynamic programming, which avoids traversing all paths (exponentially many in the size of the diagram in the worst case). For all quantum gates, there are algorithms to update the QMDD accordingly and measurement is also supported (even efficiently). Therefore, QMDDs can simulate any quantum circuit, although they

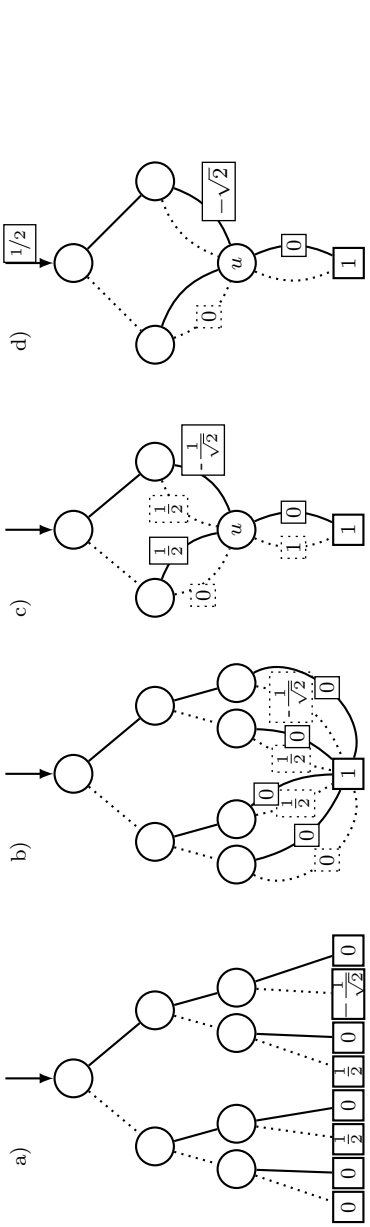


Figure 3.2: Different decision diagrams representing the 3-qubit state $[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0]^\top$, evolving into a QMDD (right). Left, (a) shows the exponential binary tree, where a node on level i represents x_i (see Equation 3.1) and its outgoing arrows $x_i = 0$ (dashed) and $x_i = 1$ (solid). The leaf contains the complex amplitude $f(x_1, x_2, x_3)$ (see Equation 3.1) for the assignment of (x_1, x_2, x_3) corresponding to the path from the root, e.g. $f(1, 1, 0) = -\frac{1}{\sqrt{2}}$. Next (b), the leaves are merged by dividing out common factors, putting these as weights (shown in boxes) on the edges going out of level-1 nodes (note in particular that we can suppress a separate 0 leaf, as $0 = 0 \cdot 1$). Then the same trick is applied to level-1 nodes in (c). In this example, all level-1 nodes v, w, s, t become *isomorphic* and can be merged into a new node u, representing the vector $|u\rangle = [1, 0]^\top$. This can be done because the level-1 nodes v, w, s, t respectively represent the vectors $[0, 0]^\top, [\frac{1}{2}, 0]^\top, [\frac{1}{2}, 0]^\top, [-\frac{1}{\sqrt{2}}, 0]^\top$, which can be written as $c \cdot |u\rangle = c \cdot [1, 0]^\top$ for respective weights $c = 0, \frac{1}{2}, \frac{1}{2}, \frac{1}{\sqrt{2}}$. Finally, (d) shows the resulting QMDD, applying the same tactic to nodes on levels 2 and 3. Note that a QMDD requires a root edge. Merging (*isomorphic*) nodes makes QMDDs succinct. By convention, unlabelled edges have label 1.

may become exponentially large (in the number of qubits) already after applying part of the gates from the circuit. The resulting simulator is *strong*, as the complete final state is computed as QMDD (and computing measurement outcome probabilities on QMDD is tractable).

Finally, we can also define the semantics of a node v recursively, overloading Dirac notation: $|v\rangle$. For convenience, we denote an edge to node v labeled with ℓ pictographically as $\xrightarrow{\ell}(v)$. Now a node v with low edge $\xrightarrow{\alpha}(v_0)$ and high edge $\xrightarrow{\beta}(v_1)$, represents the state: $|v\rangle \triangleq \alpha\,|0\rangle \otimes |v_0\rangle + \beta\,|1\rangle \otimes |v_1\rangle$, where in the base case $|\boxed{1}\rangle \triangleq 1$ as defined above. We later define LIMDD semantics similarly.

3.2.2 Pauli operators and stabilizer states

In contrast to decision diagrams, the *stabilizer formalism* [136] forms a classically simulatable subset of quantum computation. Instead of explicitly representing the (exponential) amplitude vector, the stabilizer formalism describes the symmetries a quantum state using so-called *stabilizers*. A unitary operator U *stabilizes* a state $|\varphi\rangle$ if $|\varphi\rangle$ is a $+1$ eigenvector of U, i.e., $U\,|\varphi\rangle = |\varphi\rangle$. The formalism considers stabilizers U made up of the single-qubit Pauli operators $\textsc{Pauli} \triangleq \{\mathbb{I}, X, Y, Z\}$ as defined below. In fact, a stabilizer is taken from the n-qubit Pauli group, defined as $\textsc{Pauli}_n \triangleq \langle \textsc{Pauli}^{\otimes n} \rangle$, i.e. it is the group generated by all n-qubit *Pauli strings* $P_n \otimes ... \otimes P_1$ with $P_i \in \textsc{Pauli}$. Here we used the notation $\langle G \rangle = H$ to denote that $G \subseteq H$ is a generator set for a group H. One can check that $\textsc{Pauli}_n = \{i^c P_n \otimes ... \otimes P_1 \mid P_1, ..., P_n \in \textsc{Pauli}, c \in \{0, 1, 2, 3\}\}$, so in particular we have $\textsc{Pauli}_1 = \{\pm P, \pm iP \mid P \in \textsc{Pauli}\}$ (the Pauli set with a factor ± 1 or $\pm i$).

$$\mathbb{I} \triangleq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, X \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Y \triangleq \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z \triangleq \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The set of Pauli stabilizers $\mathrm{Stab}(|\varphi\rangle) \subset \textsc{Pauli}_n$ of an n-qubit quantum state $|\varphi\rangle$ necessarily forms a subgroup of $\textsc{Pauli}_n$, since the identity operator $\mathbb{I}^{\otimes n}$ is a stabilizer of any n-qubit state and moreover if U and V stabilize $|\varphi\rangle$, then so do UV, VU and U^{-1}. Furthermore, any Pauli stabilizer group G is abelian, i.e. $A, B \in G$ implies $AB = BA$. The reason for this is that elements of $\textsc{Pauli}_n$ either commute ($AB = BA$) or anticommute ($AB = -BA$) under multiplication and anticommuting elements can never be stabilizers of the same state $|\varphi\rangle$, because if $A, B \in \mathrm{Stab}(|\varphi\rangle)$ and $AB = -BA$

then $|\varphi\rangle = AB\,|\varphi\rangle = -(BA)\,|\varphi\rangle = -\,|\varphi\rangle$, a contradiction. Finally, note that $-\mathbb{I}^{\otimes n}$ can never be a stabilizer. In fact, these conditions are necessary and and sufficient: the class of abelian subgroups G of $\textsc{Pauli}_n$, not containing $-\mathbb{I}^{\otimes n}$, are precisely all n-qubit stabilizer groups. For clarity, we adopt the convention that we denote Pauli strings *without phase* using the symbols $P, Q, R, \ldots$ and we use the symbols $A, B, C, \ldots$ for Pauli operators including phase; e.g., we may write $A = \lambda P$. The phase λ of any stabilizer $\lambda P \in \textsc{Pauli}$ can only be $\lambda = \pm 1$, derived as

$$\forall \lambda P \in \mathrm{Stab}(|\varphi\rangle): \quad |\varphi\rangle = (\lambda P)\,|\varphi\rangle = (\lambda P)^2\,|\varphi\rangle = \lambda^2 \mathbb{I}\,|\varphi\rangle = \lambda^2\,|\varphi\rangle \quad \Longrightarrow \quad \lambda = \pm 1. \tag{3.2}$$

The number of generators k for a n-qubit stabilizer group S can range from 1 to n, and S has 2^k elements. If $k = n$, then there is only a single quantum state $|\varphi\rangle$ (a single vector up to complex scalar) which is stabilized by S; such a state is called a *stabilizer state*. Equivalently, $|\varphi\rangle = C\,|0\rangle^{\otimes n}$ where C is a circuit composed of only Clifford unitaries, a group generated by the Clifford gates:

$$\text{(Hadamard gate)} \quad H \triangleq \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \text{(phase gate)} \quad S \triangleq \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}, \tag{3.3}$$

$$CNOT \triangleq \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix}. \tag{3.4}$$

In the stabilizer formalism, an n-qubit stabilizer state is succinctly represented through n independent generators of its stabilizer group, each of which is represented by $\mathcal{O}(n)$ bits to encode the Pauli string (plus factor), yielding $\mathcal{O}(n^2)$ bits in total. Examples of (generators of) stabilizer groups are $\langle Z\rangle$ for $|0\rangle$ and $\langle X \otimes X, Z \otimes Z\rangle$ for $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Updating a stabilizer state's generators after application of a Clifford gate or a single-qubit computational-basis measurement can be done in polynomial time in n [3, 136]. Various efficient algorithms exist for manipulating stabilizer (sub)groups S, including testing membership (is $A \in \textsc{Pauli}_n$ a member of S?) and finding a generating set of the intersection of two stabilizer (sub)groups. These algorithms predominantly use standard linear algebra, e.g., Gauss-Jordan elimination, as described in Sec. 3.5.2 in detail.

In this work, we also consider states which are not stabilizer states and which therefore have a nonmaximal stabilizer group (i.e. $< n$ generators). To emphasize that a stabilizer group need not be maximal, i.e. it is a subgroup of maximal stabilizer groups, we will use the term *stabilizer subgroup*. Such objects are also studied in the

context of simulating mixed states [24] and quantum error correction [135]. Examples of stabilizer subgroups are $\{\mathbb{I}\}$ for $\frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)$, $\langle -Z\rangle$ for $|1\rangle$ and $\langle X \otimes X\rangle$ for $\frac{1}{\sqrt{6}}(|00\rangle + |11\rangle) + \frac{1}{\sqrt{3}}(|01\rangle + |10\rangle)$. In contrast to stabilizer states, in general a state is not uniquely identified by its stabilizer subgroup.

Graph states on n qubits are the output states of circuits with input state $\frac{1}{2^{n/2}}(|0\rangle + |1\rangle)^{\otimes n}$ followed by only CZ $\triangleq |00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 10| - |11\rangle\langle 11|$ gates, and form a strict subset of all stabilizer states that is also important in error correction and measurement-based quantum computing [153]. By the (two-dimensional) cluster state on n^2 qubits, we mean the graph state whose graph is the $n \times n$ grid.

Given a vector space $V \subseteq \{0,1\}^n$ and a length-n bitstring s, the corresponding *coset state* is $\frac{1}{\sqrt{|V|}} \sum_{x \in V} |x + s\rangle$ where '+' denotes bitwise xor-ing [1]. Each coset state is a stabilizer state.

Stabilizer decomposition-based methods [61, 62, 64, 165, 184, 185] extend the stabilizer formalism to families of Clifford circuits with arbitrary input states $|\varphi_n\rangle$, enabling the simulation of universal quantum computation [63]. By decomposing the n-qubit state $|\varphi_n\rangle$ as linear combination of χ stabilizer states followed by simulating the circuit on each of the χ stabilizer states, the measurement outcomes can be computed in time $\mathcal{O}(\chi^2 \cdot \text{poly}(n))$, where the least χ is referred to as the *stabilizer rank* of $|\varphi_n\rangle$. Therefore, stabilizer-rank based methods are efficient for any family of input states $|\varphi_n\rangle$ whose stabilizer rank grows polynomially in n.

A specific method for obtaining a stabilizer decomposition of the output state of an n-qubit circuit is by rewriting the circuit into Clifford gates and $T = |0\rangle\langle 0| + e^{i\pi/4}|1\rangle\langle 1|$ gates (a universal gate set). Next, each of the T gates can be converted into an ancilla qubit initialized to the state $T|+\rangle$ where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$; thus, an n-qubit circuit containing t T gates will be converted into an $n + t$-qubit Clifford circuit with input state $|\varphi\rangle = |0\rangle^{\otimes n} \otimes (T|+\rangle)^{\otimes t}$ [64]. We will refer to the resulting *specific* stabilizer-rank based simulation method as the 'Clifford + T simulator,' whose simulation runtime scales with $\chi_t = \chi((T|+\rangle)^{\otimes t})$, the number of stabilizer states in the decomposition of $|\varphi\rangle$. Trivially, we have $\chi_t \leq 2^t$, and although recent work [62, 64] has found decompositions smaller than 2^t terms based on weak simulation methods, the scaling of χ_t remains exponential in t. We emphasize that the Clifford + T decomposition is not necessarily optimal, in the sense that the intermediate states of the circuit might have lower stabilizer rank than $|T\rangle^{\otimes t}$ does. Consequently, if a given circuit contains $t = \Omega(n)$ T-gates, then the Clifford + T simulator requires exponential

time (in n) for simulating this n-qubit circuit, even if there exist polynomially-large stabilizer decompositions of each of the circuit's intermediate and output states (i.e., in principle, there might exist another stabilizer rank-based simulator that can simulate this circuit efficiently).

3.2.3 Matrix product states

Representing quantum states as *matrix product states* (MPS) has proven successful for solving a large range of many-body physics problems [262,346]. For qubits, an n-qubit MPS M is formally defined as a series of $2n$ matrices $A_k^x \in \mathbb{C}^{D_k \times D_{k+1}}$ where $k \in [n], x \in \{0,1\}, D_k \in \mathbb{N}_{\geq 1}$ and $D_1 = D_{n+1} = 1$. Here, D_{k+1} is the matrix dimension over the k-th *bond*. The interpretation $|M\rangle$ is determined as $\langle x_1 x_2 ... x_n | M \rangle = A_1^{x_1} A_2^{x_2} \cdots A_n^{x_n}$ for $x_1, ..., x_n \in \{0,1\}$. If the bond dimension may scale exponentially in the number of qubits, any family of quantum states can be represented exactly by an MPS.

The *Schmidt rank* of a state $|\varphi\rangle$ on n qubits, relative to a bipartition of the qubits into two sets A and B, is the smallest integer $m \geq 1$ such that $|\varphi\rangle$ can be expressed as the superposition $|\varphi\rangle = \sum_{j=1}^{m} c_j |a_j\rangle_A |b_j\rangle_B$ for complex coefficients c_j, where the states $|a_j\rangle_A$ ($|b_j\rangle_B$) form an orthonormal basis for the Hilbert space of the A register (B register). The relation with MPS is that the maximum Schmidt rank with respect to any bipartition $A = \{x_1, \ldots, x_k\}, B = \{x_k + 1, \ldots, x_n\}$ is precisely the smallest possible bond dimension D_{k+1} required to exactly express a state in MPS.

Vidal [335] showed that MPS-based circuit simulation is possible in time $\mathcal{O}(n \cdot \mathsf{poly}(\chi))$ per elementary operation, where n is number of qubits and χ the maximum Schmidt rank for all intermediate states computed.

3.3 Local Invertible Map Decision Diagrams

Sec. 3.3.1 introduces a LIMDD definition parameterized with different local operations. We mainly consider the Pauli-LIMDD and refer to it simply as LIMDD. We show how LIMDDs generalize QMDDs and can represent arbitrary quantum states, normalized or not. We then use this definition in Sec. 3.3.2 to show how LIMDDs succinctly —i.e., in polynomial space— represent graph states (in particular cluster states), coset states and, more generally, stabilizer states. On the other hand, QMDDs and MPS require

exponential size to represent two-dimensional cluster states.

We translate this exponential advantage in quantum state representation to (universal and strong) quantum circuit simulation in Sec. 3.3.3 by giving algorithms to update and query the LIMDD data structure. These *LIMDD manipulation algorithms* take a LIMDD φ, representing some state $|\varphi\rangle$, and return another LIMDD ψ that represents the state $|\psi\rangle = U|\varphi\rangle$ for standard gates U and also for arbitrary unitaries U (by preparing U in LIMDD form first; we show how). The measurement algorithm we give returns the outcome in linear time in size of the LIMDD representation of the quantum state.

For many quantum operations, we show that our manipulation algorithms are efficient on all quantum states, i.e., take polynomial time in the size of the LIMDD representation of the state. Algorithms for certain other operations are efficient for certain classes of states, e.g., all Clifford gates can be applied in polynomial time to a LIMDD representing a stabilizer state. We show that LIMDDs can be exponentially faster than QMDDs, while they are never slower by more than a multiplicative factor $\mathcal{O}(n^3)$. These algorithms use a *canonical* form of LIMDDs, such that for each state there is a *unique* LIMDD. We defer this subject to Section 3.4, which introduces *reduced* LIMDDs and efficient algorithms to compute them.

With these algorithms, a quantum circuit simulator can be engineered by applying the circuit's gates one by one on the representation of the state as LIMDD. Prop. 3.1 provides the bottom line of this section by comparing simulator runtimes. In Sec. 3.3.4, we prove Prop. 3.1.

Proposition 3.1. Let $\text{QSim}_C^{\text{Clifford}+T}$ denote the runtime of the Clifford + T simulator on circuit C (allowing for weak simulation as in [62]). Let QSim_C^D denote the runtime of strong simulation of circuit C using method $D = (\text{Pauli}-)$LIMDD, QMDD, QMDD $\cup$ Stab, MPS, QMDD $\cup$ Stab.* Here, the latter is an (imaginary) ideal combination of QMDD (not tractable for all Clifford circuits) and the stabilizer formalism (tractable for Clifford circuits), i.e., one that always inherits the best worst-case runtime from either method.
The following holds, where Ω^* discards polynomial factors, i.e., $\Omega^*(f(n)) \triangleq \Omega(n^{\mathcal{O}(1)} f(n))$.

There is a family of circuits C such that:

*We are not aware of any (potentially better) weak D-based simulation approaches and do not consider them.

1. LIMDD is exponentially faster than Clifford + T: $\mathrm{QSIM}_C^{\text{Clifford + }T} = \Omega^*(2^n \cdot \mathrm{QSIM}_C^{\mathsf{LIMDD}})$,[†]
2. LIMDD is exponentially faster than MPS: $\mathrm{QSIM}_C^{\mathrm{MPS}} = \Omega^*(2^n \cdot \mathrm{QSIM}_C^{\mathsf{LIMDD}})$, and
3. LIMDD is exponentially faster than QMDD: $\mathrm{QSIM}_C^{\mathsf{QMDD}} = \Omega^*(2^n \cdot \mathrm{QSIM}_C^{\mathsf{LIMDD}})$.
4. For all C, LIMDD is at worst cubically slower than QMDD: $\mathrm{QSIM}_C^{\mathsf{LIMDD}} = \mathcal{O}(n^3 \cdot \mathrm{QSIM}_C^{\mathsf{QMDD}})$.
5. Item 3 and 4 hold when replacing QMDD with QMDD $\cup$ Stab.

3.3.1 The LIMDD data structure

Where QMDDs only merge nodes representing the same complex vector up to a constant factor, the LIMDD data structure goes further by also merging nodes that are equivalent up to local operations, called Local Invertible Maps (LIMs) (see Definition 3.1). As a result, LIMDDs can be exponentially more succinct than QMDDs, for example in the case of stabilizer states (see Sec. 3.3.2). We will call nodes which are equivalent under LIMs, *(LIM-) isomorphic*. This definition generalizes SLOCC equivalence (Stochastic Local Operations and Classical Communication); if we choose the parameter G to be the linear group, then the two notions coincide (see [107, App. A] and [39, 82]).

Definition 3.1 (G-LIM, G-Isomorphism)**.** An n-qubit G-Local Invertible Map (LIM) is an operator $\mathcal{O}$ of the form $\mathcal{O} = \lambda \mathcal{O}_n \otimes \cdots \otimes \mathcal{O}_1$, where G is a group of invertible 2×2 matrices, $\mathcal{O}_i \in G$ and $\lambda \in \mathbb{C} \setminus \{0\}$. A *$G$-isomorphism* between two n-qubit quantum states $|\varphi\rangle, |\psi\rangle$ is a LIM $\mathcal{O}$ such that $\mathcal{O}|\varphi\rangle = |\psi\rangle$, denoted $|\varphi\rangle \simeq_G |\psi\rangle$. Note that G-isomorphism is an equivalence relation.

We define $\mathrm{PAULILIM}_n \triangleq \langle \mathrm{PAULI} \rangle$-LIM, i.e., the group of Pauli operators $P \in \mathrm{PAULI}_n$ with arbitrary complex factor $\lambda \in \mathbb{C} \setminus \{0\}$ (λ can absorb the factor $\gamma = \pm 1, \pm i$ in $P = \gamma P_n \otimes ... \otimes P_1$. Note $\lambda = \pm 1$ still for $\mathrm{PAULILIM}_n$ operators which are stabilizers, by eq. (3.2)).

Before we give the formal definition of LIMDDs in Definition 3.2, we give a motivating example in Figure 3.3, which uses $\langle X, Y, Z, T \rangle$-LIMs to demonstrate how the use of

[†]Assuming the exponential time hypothesis (ETH). See Sec. 3.3.4.3 for details.

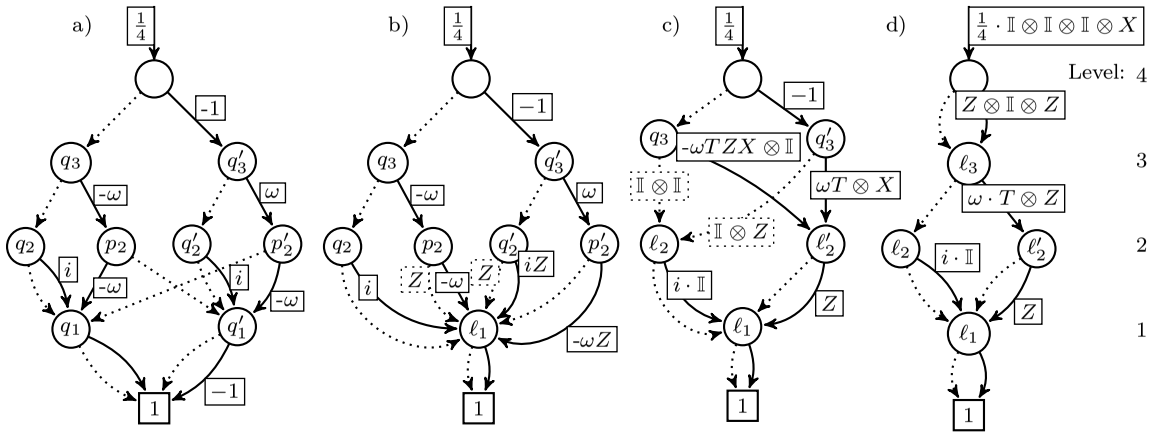


Figure 3.3: A QMDD (a) representing the state $\frac{1}{4}[1,1,i,i,-\omega,\omega,i,i,-1,1,-i,i,-\omega,-\omega,i,-i]^\top$ with $\omega = e^{i\pi/4}$, evolving into a LIMDD (d). As in Figure 3.2, diagram nodes are horizontally ordered in 'levels' with qubit indices $4,3,2,1$. Low edges are dashed, high edges solid. See the text for an explanation.
By convention, unlabelled edges have label 1 (for QMDD) or $\mathbb{I}^{\otimes k}$ (for LIMDD nodes at level k).

isomorphisms can yield small diagrams for a four-qubit state. This figure shows how to merge nodes in four steps, shown in subfigures (a)-(d), starting with a large QMDD (a) and ending with a small LIMDD (d). In the QMDD (a), the nodes labeled q_1 and q_1' represent the single-qubit states $|q_1\rangle = [1,1]^\top$ and $|q_1'\rangle = [1,-1]^\top$, respectively. By noticing that these two vectors are related via $|q_1'\rangle = Z\,|q_1\rangle$, we merge nodes q_1, q_1' into node ℓ_1 in (b), storing the isomorphism Z on all incoming edges that previously pointed to q_1'. From step (b) to (c), we first merge q_2, q_2' into ℓ_2, observing that $|q_2'\rangle = \mathbb{I} \otimes Z\,|q_2\rangle$. Second, we create a node ℓ_2' such that $|p_2\rangle = TZX \otimes I\,|\ell_2'\rangle$ and $|p_2'\rangle = T \otimes X\,|\ell_2'\rangle$. So we can merge nodes p_2, p_2' into ℓ_2', placing these isomorphisms on the respective edges. To go from (c) to (d), we merge nodes q_3, q_3' into node ℓ_3 by noticing that $|q_3'\rangle = (Z \otimes \mathbb{I} \otimes Z)\,|q_3\rangle$. This isomorphism $Z \otimes \mathbb{I} \otimes Z$ is stored on the high edge out of the root node. We have $|q_3\rangle = \mathbb{I} \otimes \mathbb{I} \otimes X\,|\ell_3\rangle$, so we propagate the isomorphism $\mathbb{I} \otimes \mathbb{I} \otimes X$ upward, and store it on the root edge. Therefore, the final LIMDD has the LIM $\frac{1}{4}\mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I} \otimes X$ on its root edge.

The resulting data structure in Figure 3.3 is a LIMDD of only six nodes instead of ten, but requires additional storage for the LIMs. Sec. 3.3.2 shows that merging isomorphic nodes sometimes leads to exponentially smaller diagrams, while the additional cost of storing the isomorphisms results only costs a linear factor of space (linear in the number of qubits).

The transformation presented above (for Figure 3.3) only considers particular choices for LIMs. For instance, it would be equally valid to select LIM $\mathbb{I} \otimes Z$ instead of $-\mathbb{I} \otimes XZ$ for mapping q_2' onto q_2. In fact, efficient algorithms to select LIMs in such a way that a canonical LIMDD is obtained are a cornerstone for the LIMDD manipulation algorithms presented in Sec. 3.3.3. Section 3.4 provides a solution for $\langle$Pauli$\rangle$-LIMs (the basis for all results presented in the current article), which is based on using the stabilizers of each node, e.g., the group generated by $\{\mathbb{I} \otimes X, Y \otimes \mathbb{I}\}$ for q_2.

Definition 3.2. An n-G-LIMDD is a rooted, directed acyclic graph (DAG) representing an n-qubit quantum state. Formally, it is a 6-tuple (Node $\cup$ {Leaf}, idx, low, high, label, e_{root}), where:

- Leaf (a sink) is a unique leaf node with qubit index idx(Leaf) $= 0$;
- Node is a set of nodes with qubit indices idx$(v) \in \{1, \ldots, n\}$ for $v \in$ Node;
- e_{root} is a root edge without source pointing to the root node $r \in$ Node with idx$(r) = n$;

- $\mathsf{low}, \mathsf{high}\colon \textsc{Node} \to \textsc{Node} \cup \{\mathsf{Leaf}\}$ indicate the low and high edge functions, respectively. We write low_v (or high_v) to obtain the edge (v, w) with $w = \mathsf{low}(v)$ (or $w = \mathsf{high}(v)$). For all $v \in \textsc{Node}$ it holds that $\mathsf{idx}(\mathsf{low}(v)) = \mathsf{idx}(\mathsf{high}(v)) = \mathsf{idx}(v) - 1$ (no qubits are skipped[‡]);
- $\mathsf{label}\colon \mathsf{low} \cup \mathsf{high} \cup \{e_{\text{root}}\} \to k\text{-}G\text{-LIM} \cup \{0\}$ is a function labeling edges $(\,.\,, w)$ with k-G-LIMs or 0, where $k = \mathsf{idx}(w)$

We will find it convenient to write $(v) \overset{A}{\cdots} (u) \overset{B}{\text{—}} (w)$ for a node u with low and high edges to nodes v and w labeled with A and B, respectively. We will also denote $\overset{A}{\text{—}}(v)$ for a (root) edge to v labeled with A. When omitting A or B, e.g., $\text{—}(v)$, the LIM should be interpreted as $\mathbb{I}^{\otimes \mathsf{idx}(v)}$.

We define the semantics of a leaf, node v and an edge e to node v by overloading the Dirac notation:

$$\begin{aligned} |\mathsf{Leaf}\rangle &\triangleq 1 \\ |e\rangle &\triangleq \mathsf{label}(e) \cdot |v\rangle \\ |v\rangle &\triangleq |0\rangle \otimes |\mathsf{low}_v\rangle + |1\rangle \otimes |\mathsf{high}_v\rangle \end{aligned}$$

It follows from this definition that a node v with $\mathsf{idx}(v) = k$ represents a quantum state on k qubits. *This state is however not necessarily normalized:* For instance, a normalized state $\alpha\,|0\rangle + \beta\,|1\rangle$, can be represented as a LIMDD $\boxed{1} \overset{\alpha}{\cdots} (v) \overset{\beta}{\text{—}} \boxed{1}$ or a LIMDD $\boxed{1} \cdots (v) \overset{\beta/\alpha}{\text{—}} \boxed{1}$ with root edge $\overset{\alpha}{\text{—}}(v)$. So the node v represents a state up to global scalar. But, in general, any scalar can be applied to the root edge, or any other edge for that matter. So LIMDDs can represent any complex vector.

The tensor product $|e_0\rangle \otimes |e_1\rangle$ of the G-LIMDDs with root edges e_0 $\overset{A}{\text{—}}(v)$ and e_1 $\overset{B}{\text{—}}(w)$ can be computed just like for QMDDs [227]: Take all edges $\overset{\alpha}{\text{—}}\boxed{1}$ pointing to the leaf in the LIMDD e_0 and replace them with edges $\overset{\alpha \cdot B}{\text{—}}(w)$ pointing to the e_1 root node w. The result is an $n + m$ level LIMDD if e_0 has n levels and e_1 has m. In addition, the LIMs C on the other edges in the LIMDD e_0 should be extended to $C \otimes \mathbb{I}^{\otimes m}$.

We can now consider various instantiations of the above general LIMDD definition for

[‡]Decision diagram definitions [7, 67, 115] often allow to skip (qubit) variables, interpreting them as 'don't cares.' We disallow this here, since it complicates definitions and proofs, while at best it yields linear size reductions [183].

different LIM groups G. A G-LIMDD with $G = \{\mathbb{I}\}$ yields precisely all QMDDs by definition, i.e., all edges labels effectively only contain scalars. As all groups G contain the identity operator $\mathbb{I}$, the universality of G-LIMDDs (i.e., all quantum states can be represented) follows from the universality of QMDDs. It also follows that any state that can efficiently be represented by QMDD, can be efficiently represented by a G-LIMDD for any G. Similarly, we can consider $\langle Z\rangle$ and $\langle X\rangle$, which are subgroups of the Pauli group, and define a $\langle Z\rangle$-LIMDD and a $\langle X\rangle$-LIMDD; instances that we will study for their relation to graph states and coset states in Sec. 3.3.2. Finally, and most importantly, $\langle$Pauli$\rangle$-LIMDDs can represent all stabilizer states in polynomial space, which is a feature that neither QMDDs nor matrix product states (MPS) posses, as shown in Sec. 3.3.2.

In what follows, we only consider $\langle X\rangle$-, $\langle Z\rangle$-, and $\langle$Pauli$\rangle$-LIMDDs, or Pauli-LIMDD for short. For Pauli-LIMDDs, we now illustrate how to find the amplitude of a computational basis state $\langle x|\psi\rangle$ for a bitstring $x \in \{0,1\}^n$ by traversing the LIMDD of the state $|\psi\rangle$ from root to leaf, as follows. Suppose that this diagram's root edge e_{root} points to node r and is labeled with the LIM $\mathsf{label}(e_{\text{root}}) = A = \lambda P_n \otimes \cdots \otimes P_1 \in \text{PauliLIM}_n$. First, we substitute $|r\rangle = |0\rangle |\mathsf{low}_r\rangle + |1\rangle |\mathsf{high}_r\rangle$, where $\mathsf{low}_r, \mathsf{high}_r$ are the low and high edges going out of r, thus obtaining $\langle x|\psi\rangle = \langle x|e_{\text{root}}\rangle = \langle x| A (|0\rangle |\mathsf{low}_r\rangle + |1\rangle |\mathsf{high}_r\rangle)$. Next, we notice that $\langle x| A = \lambda(\langle x_n| P_n) \otimes \cdots \otimes (\langle x_1| P_1) = \gamma \langle y|$ for some $\gamma \in \mathbb{C}$ and a computational basis state $\langle y|$. Therefore, letting $y' = y_{n-1} \dots y_1$, it suffices to compute $\langle y_n| \langle y'| (|0\rangle |\mathsf{low}_r\rangle + |1\rangle |\mathsf{high}_r\rangle)$, which reduces to computing either $\langle y'|\mathsf{low}_r\rangle$ if $y_n = 0$, or $\langle y'|\mathsf{high}_r\rangle$ if $y_n = 1$. By applying this simple rule repeatedly, one walks from the root to the leaf, encountering one node on each level. The amplitude $\langle x|\psi\rangle$ is then found by multiplying together the scalars γ found along this path. Algorithm 1 formalizes this. Its runtime is $\mathcal{O}(n^2)$.

Algorithm 1 Read the amplitude for basis state $|x_n...x_1\rangle$ from n-qubit state $|e\rangle = A \cdot |v\rangle$ with $A = \lambda P_n \otimes ... \otimes P_1 \in \text{PauliLIM}_n$.

procedure ReadAmplitude(Edge e $\xrightarrow{A}$ (v), $x_n, ..., x_1 \in \{0,1\}$ **with** $n = \mathsf{idx}(v)$)
  if $n = 0$ **then return** λ
  $\gamma \langle y_n...y_1| := \langle x_n...x_1| \lambda P_n \otimes ... \otimes P_1$ ▷ $\mathcal{O}(n)$-computable LIM operation
  if $y_n = 0$ **then** ▷ $y_n = 0$
    return $\gamma \cdot$ ReadAmplitude($\mathsf{low}_v, y_{n-1}, ..., y_1$)
  else ▷ $y_n = 1$
    return $\gamma \cdot$ ReadAmplitude($\mathsf{high}_v, y_{n-1}, ..., y_1$)

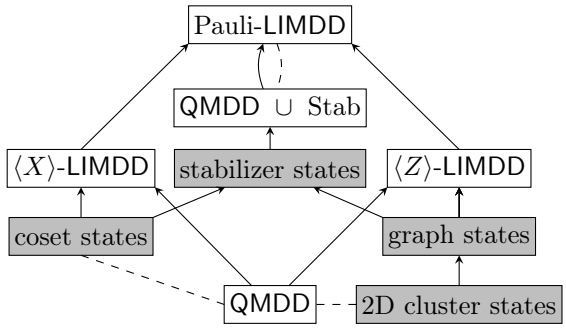


Figure 3.4: Relations between non-universal classes of quantum states (gray) and decision diagrams, where we consider a diagram as the set of states that it can represent in polynomial size. Solid arrows denote set inclusion. Dashed arrows $D_1 \dashrightarrow D_2$ signify an exponential separation between two classes, i.e., some quantum states have polynomial-size representation in D_1, but only exponential-size in D_2.
By transitivity, QMDD is exponentially separated from all representations (not drawn for clarity).

3.3.2 Succinctness of LIMDDs

Succinctness is crucial for efficient simulation, as we show later. In this section, we show exponential advantages for representing states with LIMDDs over two other state-of-the-art data structures: QMDDs and Matrix Product States (MPS) [262,346]. Specifically, QMDDs and MPS require exponential space in the number of qubits to represent specific stabilizer states called (two-dimensional) cluster states. We also show that an ad-hoc combination of QMDD with the stabilizer formalism still requires exponential space for 'pseudo-cluster states.' These results are visualized in Figure 3.1.

3.3.2.1 LIMDDs are exponentially more succinct than QMDDs (union stabilizer states)

Figure 3.4 visualizes succinctness relations between different quantum state representations, as proved in Prop. 3.2. In particular, G-LIMDDs with $G = \langle \text{PAULI} \rangle$ can be exponentially more succinct than QMDDs, and retain this exponential advantage even with $G = \langle Z \rangle, \langle X \rangle$. In Corollary 3.1, we show the strongest result, namely that LIMDDs are also more succinct than the union of QMDDs and stabilizer states, written QMDD ∪ Stab, which can be thought of a structure that switches between QMDD and the stabilizer formalism depending on its content (stabilizer or non-stabilizer

state). This demonstrates that ad-hoc combinations of existing formalisms do not make LIMDDs obsolete.

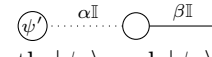
Proposition 3.2. The inclusions and separations in Figure 3.4 hold.

Proof. The inclusions between the sets of states shown in gray are well known [1, 153]. The inclusions between decision diagrams hold because, e.g., a QMDD is a G-LIMDD with $G = \{\mathbb{I}\}$, i.e., each label is of the form $\lambda\mathbb{I}_n$ with $\lambda \in \mathbb{C}$, as discussed in Sec. 3.3.1. The relations between coset, graph, stabilizer states and G-LIMDD with $G = \langle X \rangle, \langle Z \rangle, \langle \text{PAULI} \rangle$ are proven in Theorem 3.1 and App. B (which also shows that poly-sized LIMDD includes QMDD $\cup$ Stab). Corollary 3.1 shows that there is family of a non-stabilizer states (with small LIMDD) for which QMDD is exponential, hence the separation between QMDD $\cup$ Stab. Theorem 3.2 shows the separation with QMDDs by demonstrating that the so-called (two-dimensional) cluster state, requires $2^{\Omega(\sqrt{n})}$ nodes as QMDD. Finally, App. B proves the same for coset states. □

Theorem 3.1 shows that any stabilizer state can be represented as a $\langle$PAULI$\rangle$-Tower LIMDD (Definition 3.3).

Definition 3.3. A n-qubit G-Tower-LIMDD, is a G-LIMDD with exactly one node on each level. Edges to nodes on level k are labeled as follows: low edges are labeled with $\mathbb{I}^{\otimes k}$, high edges with $P \in G^{\otimes k} \cup \{0\}$ and the root edge is labeled with $\lambda \cdot P$ with $P \in G^{\otimes k}$ and $\lambda \in \mathbb{C} \setminus \{0\}$ (i.e., in contrast to high edges, the root edge can have an arbitrary scalar). Figure 3.6 depicts a n-qubit G-Tower LIMDD.

Theorem 3.1. Let $n > 0$. Each n-qubit stabilizer state is represented up to normalization by a $\langle$PAULI$\rangle$-Tower LIMDDs of Definition 3.3, e.g., where the scalars λ of the PAULILIMs λP on high edges are restricted as $\lambda \in \{0, \pm 1, \pm i\}$. Conversely, every such LIMDD represents a stabilizer state.

Proof sketch of Theorem 3.1. (Full proof in App. B) The $n = 1$ case: the six single-qubit states $|0\rangle, |1\rangle, |0\rangle \pm |1\rangle$ and $|0\rangle \pm i\,|1\rangle$ are all represented by a $\langle$PAULI$\rangle$-Tower LIMDD with a single node on top of the leaf. The induction step: Let $|\psi\rangle$ be an n-qubit stabilizer state. First, consider the case that $|\psi\rangle = |a\rangle\,|\psi'\rangle$ where $|a\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ (with $\alpha, \beta \in \{0, \pm 1, \pm i\}$) and $|\psi'\rangle$ are stabilizer states on respectively 1 and $n-1$ qubits. Then $|\psi\rangle$ is represented by the $\langle$PAULI$\rangle$-Tower-LIMDD 

. In the remaining case, $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle\,|\psi_0\rangle + |1\rangle\,|\psi_1\rangle)$, where both $|\psi_0\rangle$ and $|\psi_1\rangle$ are

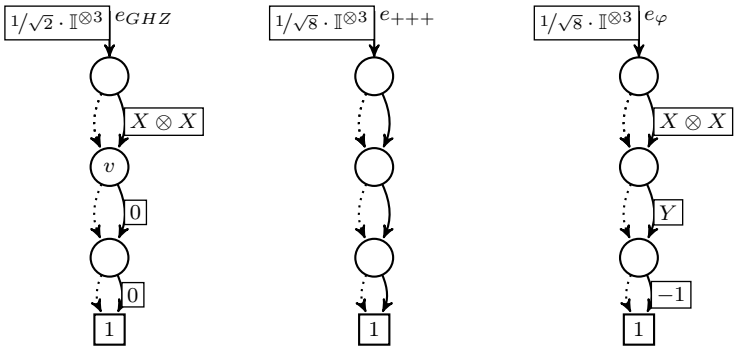


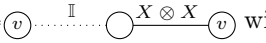
Figure 3.5: Figure 3.5: Example ⟨Pauli⟩-Tower LIMDDs for three stabilizer states: the GHZ state $|e_{GHZ}\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, for $|e_{+++}\rangle = |+++\rangle$ where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and the state $|e_{\varphi}\rangle = \frac{1}{\sqrt{8}}(|000\rangle - |001\rangle + i\,|010\rangle + i\,|011\rangle + i\,|100\rangle + i\,|101\rangle - |110\rangle + |111\rangle)$ with stabilizer group generators $\{X \otimes X \otimes X, -Z \otimes Z \otimes X, \mathbb{I} \otimes Y \otimes Z\}$.

stabilizer states. Moreover, since $|\psi\rangle$ is a stabilizer state, there is always a set of single-qubit Pauli gates $P_1, \ldots, P_n$ and a $\lambda \in \{\pm 1, \pm i\}$ such that $|\psi_1\rangle = \lambda P_n \otimes \cdots \otimes P_1 |\psi_0\rangle$. That is, in our terminology, the states $|\psi_0\rangle$ and $|\psi_1\rangle$ are *isomorphic*. Hence $|\psi\rangle$ can be written as

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left[|0\rangle |\psi_0\rangle + \lambda |1\rangle \otimes (P_n \otimes \cdots \otimes P_1 |\psi_0\rangle)\right] \tag{3.5}$$

Hence $|\psi\rangle$ is represented by the Tower Pauli-LIMDD 

. In both cases, $|\psi'\rangle$ is represented by a Tower Pauli-LIMDDs (up to normalization) by the induction hypothesis. □

We stress that obtaining the LIMs for the Pauli Tower-LIMDD of a stabilizer state is not immediate from the stabilizer generators; specifically, the edge labels in the Pauli-LIMDD are not directly the stabilizers of the state. For example, the GHZ state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ is represented by $|e_{GHZ}\rangle = \frac{1}{\sqrt{2}}$ 

with $|v\rangle = |00\rangle$ in Figure 3.5, but $X \otimes X$ is not a stabilizer of $|00\rangle$. Nonetheless, Theorem 3.1 implicitly contains an algorithm that constructs a ⟨Pauli⟩-Tower LIMDD stabilizer state. Section 3.4 also provides the inverse construction, which we use to make LIMDDs (representing any quantum state) canonical in time $\mathcal{O}(mn^3)$ (using Algorithm 3).

We also note that Theorem 3.1 demonstrates that for any n-qubit stabilizer state $|\varphi\rangle$,

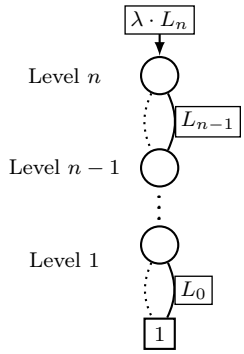


Figure 3.6: Figure 3.6: An n-qubit G-Tower LIMDD. We let $L_i \in G^{\otimes i} \cup \{0\}$ and $\lambda \in \mathbb{C} \setminus \{0\}$ (only root edges have an arbitrary scalar).

the $(n-1)$-qubit states $(\langle 0| \otimes \mathbb{I}_{2^{n-1}}) |\varphi\rangle$ and $(\langle 1| \otimes \mathbb{I}_{2^{n-1}}) |\varphi\rangle$ are not only stabilizer states, but also PauliLIM-isomorphic. While we believe this fact is known in the community,[§] we have not found this statement written down explicitly in the literature. More importantly for this work, to the best of our knowledge, the resulting recursive structure (which DDs are) has not yet been exploited in the context of classical simulation.

Next, Theorem 3.2 shows the separation with QMDDs by demonstrating that the so-called (two-dimensional) cluster state, requires $2^{\Omega(\sqrt{n})}$ nodes as QMDD. Corollary 3.1 shows that a trivial combination with stabilizer formalism does not solve this issue.

Theorem 3.2. Denote by $|G_n\rangle$ the two-dimensional cluster state, defined as a graph state on the $n \times n$ lattice. Each QMDD representing $|G_n\rangle$ has at least $2^{\lfloor n/12 \rfloor}$ nodes.

Proof sketch. Consider a partition of the vertices of the $n \times n$ lattice into two sets S and T of size $\frac{1}{2}n^2$, corresponding to the first $\frac{1}{2}n^2$ qubits under some variable order. Then there are at least $\lfloor n/3 \rfloor$ vertices in S that are adjacent to a vertex in T [204, Th. 11]. Because the degree of the vertices is small, many vertices on this boundary are not connected and therefore influence the amplitude function independently of one another. From this independence, it follows that, for any variable order, the partial assignments $\vec{a} \in \{0,1\}^{\frac{1}{2}n^2}$ induce $2^{\lfloor n/12 \rfloor}$ different subfunctions $f_{\vec{a}}$, where $f\colon \{0,1\}^{n^2} \to \mathbb{C}$ is the

[§]For instance, this fact can be observed (excluding global scalars) by executing the original algorithm for simulating single-qubit computational-basis measurement on the first qubit, as observed in [136]. Similarly, the characterization in Prop. 3.2 of $\langle Z \rangle$-Tower-LIMDDs as representing precisely the graph states, is immediate by defining graph states recursively (see App. B). The fact that $\langle X \rangle$-Tower LIMDDs represent coset states is less evident and requires a separate proof, which we also give in App. B.

amplitude function of $|G_n\rangle$. The lemma follows by noting that a QMDD has a single node per unique subfunction modulo phase. For details see App. A. □

Corollary 3.1 (Exponential separation between Pauli-LIMDD versus QMDD union stabilizer states)**.** There is a family of non-stabilizer states, which we call *pseudo cluster states*, that have polynomial-size Pauli-LIMDD but exponential-size QMDDs representation.

Proof. Consider the pseudo cluster state $|\varphi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle) \otimes |G_n\rangle$ where $|G_n\rangle$ is the graph state on the $n \times n$ grid. This is not a stabilizer state, because each computational-basis coefficient of a stabilizer state is of the form $z \cdot \frac{1}{\sqrt{2}^k}$ for $z \in \{\pm 1, \pm i\}$ and some integer $k \geq 1$ [325], while $\langle 1| \otimes \langle 0|^{\otimes n^2} |\varphi\rangle = e^{i\pi/4} \cdot \left(\frac{1}{\sqrt{2}}\right)^{n^2+1}$ is not of this form. Its canonical QMDD and Pauli-LIMDD have root nodes $G_n \xrightarrow{1} e^{i\pi/4} G_n$ and $G_n \xrightarrow{\mathbb{I}} e^{i\pi/4}\mathbb{I} G_n$, where the respective diagram for G_n is exponentially large (Theorem 3.2) and polynomially small (Theorem 3.1). □

3.3.2.2 LIMDDs are exponentially more succinct than matrix product states

Theorem 3.3 states that matrix product states (MPS) require large bond dimension for representing the two-dimensional cluster states, which follows directly from the well-known results that these states have large Schmidt rank.

Theorem 3.3. To represent the graph state on the $n \times n$ grid (the two-dimensional cluster state on n^2 qubits), an MPS requires bond dimension $2^{\Omega(n)}$.

Proof. Van den Nest et al. [323] consider spanning trees over the complete graph where each node corresponds to a qubit and define the Schmidt-rank width: the largest encountered base-2 logarithm of the Schmidt rank between the two connected components resulting from removing an edge from the spanning tree, minimized over all possible spanning trees. It then follows from the relation between bond dimension and Schmidt rank (see Section 3.2) that any quantum state with Schmidt-rank width w requires bond dimension 2^w for representation by an MPS. Van den Nest et al. also showed that for graph states, the Schmidt-rank width equals the so-called rank width of the graph, which for $n \times n$ grid graphs was shown to equal $n - 1$ by Jelinek [168]. This proves the theorem. □

In contrast, the Pauli-LIMDD efficiently represents cluster states, and more generally all stabilizer states (Theorem 3.1).

3.3.3 Pauli-LIMDD manipulation algorithms for simulation of quantum computing

In this section, we give all algorithms that are necessary to simulate a quantum circuit with Pauli-LIMDDs (referred to simply as LIMDD from now on). We provide algorithms which update the LIMDD after an arbitrary gate and after a single-qubit measurement in the computational basis. In addition, we give efficient specialized algorithms for applying a Clifford gate to a stabilizer state (represented by a ⟨Pauli⟩-Tower LIMDD) and computing a measurement outcome. We also show that many (Clifford) gates can in fact be applied to an arbitrary state in polynomial time. Table 3.1 provides an overview of the LIMDD algorithms and their complexities compared to QMDDs.

Central to the speed of many DD algorithms is keeping the diagram canonical throughout the computation. Recall from Sec. 3.3.1, that a G-LIMDD can merge isomorphic nodes $v \simeq_G w$, i.e., if there exists a G-LIM C such that $|w\rangle = C\,|v\rangle$. To achieve this, we require a 'MakeEdge' subroutine which, given the node v_0 with edges A to w and B to v_1, returns C to v with $C\,|v\rangle = |w\rangle$, where v is the unique, canonical node in the diagram that is G-isomorphic to node w. Sec. 3.4.2 provides a $\mathcal{O}(n^3)$ MakeEdge algorithm

Table 3.1: Worst-case complexity of currently *best-known algorithms* for applying specific operations, in terms of the size of the input diagram size m (i.e., the number of nodes in the DD) and the number of qubits n. Although addition (Add) of quantum states is not, strictly speaking, a quantum operation, we include it because it is a subroutine of gate application. Note that several of the LIMDD algorithms invoke MakeEdge and therefore inherit its cubic complexity (as a factor).

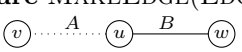
Operation \ input:	**QMDD**	**LIMDD**	**Section**
Single $\|0\rangle$ / $\|1\rangle$-basis measurement	$\mathcal{O}(m)$	$\mathcal{O}(m)$	Sec. 3.3.3.1
Single Pauli gate	$\mathcal{O}(m)$	$\mathcal{O}(1)$	Sec. 3.3.3.2
Single Hadamard gate / Add()	$\mathcal{O}(2^n)$ ¶	$\mathcal{O}(n^3 2^n)$ ¶	Sec. 3.3.3.2
Clifford gate on stabilizer state	$\mathcal{O}(2^n)$	$\mathcal{O}(n^4)$	Sec. 3.3.3.4
Multi-qubit gate	$\mathcal{O}(4^n)$	$\mathcal{O}(n^3 4^n)$	Sec. 3.3.3.3
MakeEdge	$\mathcal{O}(1)$	$\mathcal{O}(n^3)$	Sec. 3.4.2
Checking state equality	$\mathcal{O}(1)$	$\mathcal{O}(n^3)$	Sec. 3.4.2.2

¶The worst-case of QMDDs and LIMDDs is caused by the vector addition introduced by the Hadamard gate [113, Table 2, +BC, +SLDD]. See Figure 3.9 for an example.

for ⟨Pauli⟩-LIMDDs satisfying this specification. For now, the reader may assume the provisional implementation in Algorithm 2, which does not yet merge LIM-isomorphic nodes and hence does not yield canonical diagrams.

In line with other existing efficient decision-diagram algorithms, we use dynamic programming in our algorithms to avoid traversing all paths (possibly exponentially many) in the LIMDD. In this approach, the decision diagram is manipulated and queried using recursive algorithms, which store intermediate results for each recursive call to avoid unnecessary recomputations. For instance, Algorithm 3 makes any LIMDD canonical using dynamic programming and the (real) $\mathcal{O}(n^3)$ MakeEdge algorithm from Sec. 3.4.2. It recursively traverses child nodes at Line 3, reconstructing the diagram bottom up in the backtrack at Line 4. By virtue of dynamic programming it visits each node only once: The table CanonicalCache: Node → Edge stores for each node its canonical counterpart as soon as it is computed at Line 4. The algorithm therefore runs in time $\mathcal{O}(n^3 m)$ where m is the number of nodes in the original diagram.

Algorithm 2 Provisionary algorithm MakeEdge for creating a new node/edge. Given two edges representing states $A|v\rangle, B|w\rangle$, it returns an edge representing the state $|0\rangle A|v\rangle + |1\rangle B|w\rangle$. The real MakeEdge algorithm (Sec. 3.4.2) returns a canonical node, assuming v, w are already canonical.

1: **procedure** MakeEdge(Edge $\xrightarrow{A}(v)$, Edge $\xrightarrow{B}(w)$)
2: $u :=$ 
3: **return** Edge  ▷ Where $k = \mathsf{idx}(u)$

Algorithm 3 Make any LIMDD canonical using MakeEdge.

1: **procedure** MakeCanonical(Edge $\xrightarrow{A}(v)$)
2: **if** $v \notin$ CanonicalCache **then** ▷ Compute result once for v and store in cache:
3: $e_0,\ e_1 :=$ MakeCanonical($\mathsf{low}(v)$), MakeCanonical($\mathsf{high}(v)$)
4: CanonicalCache$[v]$:= MakeEdge(e_0, e_1)
5: **return** $A \cdot$ CanonicalCache$[v]$ ▷ Retrieve result from cache

This recursive algorithmic structure that uses dynamic programming and reconstructs the diagram in the backtrack, is typical for all DD manipulation algorithms. Note that constant-time cache lookups (using a hash table) therefore require the canonical nodes produced by MakeEdge. LIMDDs additionally require the addition of LIMs to the caches; Sec. 3.3.3.3 shows how we do this.

Finally, in this section, we often decompose LIMS using $A = \lambda P_n \otimes P'$. Here $\lambda \in \mathbb{C}$ is

a non-zero scalar, P' a Pauli string and $P_n \in \{\mathbb{I}, X, Z, Y\} = \text{PAULI}$. Our algorithms will use the Follow procedure from Algorithm 4 to easily navigate diagrams according to edge semantics. Provided with a bit string $x_n...x_1$, the procedure is the same as ReadAmplitude. If however fewer bits are supplied, it returns a LIMDD root edge representing a subvector. For instance, the subvector for $|10\rangle$ of the LIMDD root edge e_r in Figure 3.3 (d) is computed by taking $|\text{follow}(10, e_r)\rangle = |\ \xrightarrow{\frac{1}{4}\mathbb{I}\otimes XZ}\ell_2\rangle = \frac{1}{4}\cdot[-1, 1, -i, i]$. So, we can specify it as $|\text{follow}(b, e)\rangle = (\langle b| \otimes \mathbb{I}^{n-\ell})\,|e\rangle$, i.e., select the bth block of size $2^{n-\ell}$ from the vector $|e\rangle$ (or rather, return a LIMDD edge representing that block).

Algorithm 4 Follow: a generalization of ReadAmplitude, returning edges.

1: **procedure** Follow(Edge $e \xrightarrow{\lambda P_n \otimes ... \otimes P_1} v$, $x_n, ..., x_k \in \{0, 1\}$ **with** $n = \mathsf{idx}(v)$ and $k \geq 1$)
2: **if** $k > n$ **then return** $\xrightarrow{\lambda P_n \otimes ... \otimes P_1} v$ ▷ End of bit string
3: $\gamma\,\langle y_n...y_k| := \langle x_n...x_k|\,\lambda P_n \otimes ... \otimes P_k$ ▷ $\mathcal{O}(n)$-computable LIM operation
4: **if** $y_n = 0$ **then** ▷ $y_n = 0$
5: **return** $\gamma \cdot \text{Follow}(\mathsf{low}_v, y_{n-1}, ..., y_k)$
6: **else** ▷ $y_n = 1$
7: **return** $\gamma \cdot \text{Follow}(\mathsf{high}_v, y_{n-1}, ..., y_k)$

3.3.3.1 Performing a measurement in the computational basis

We discuss algorithms for measuring, sampling and updating after measurement of the top qubit. App. C gives general algorithms with the same worst-case runtimes.

The procedure MeasurementProbability in Algorithm 5 computes the probability p of observing the outcome $|0\rangle$ for state $|e\rangle$. If the quantum state can be written as $|e\rangle = |0\rangle\,|e_0\rangle + |1\rangle\,|e_1\rangle$, then the probability is $p = \langle e_0|e_0\rangle\,/\,\langle e|e\rangle$, where we have $\langle e|e\rangle = \langle e_0|e_0\rangle + \langle e_1|e_1\rangle$. Hence we compute the squared norms of $e_x = \text{follow}(x, e)$ using the SquaredNorm subroutine. The total runtime is dominated by the subroutine SquaredNorm, which computes the quantity $\langle e|e\rangle$ given a LIMDD edge $e = \xrightarrow{\lambda P} v$ by traversing the entire LIMDD. We have $\langle e|e\rangle = |\lambda|^2\,\langle v|\,P^\dagger P\,|v\rangle = |\lambda|^2\,\langle v|v\rangle$, because $P^\dagger P = \mathbb{I}$ for Pauli matrices. Therefore, to this end, it computes the squared norm of $|v\rangle$. Since $\langle v|v\rangle = \langle \mathsf{low}_v|\mathsf{low}_v\rangle + \langle \mathsf{high}_v|\mathsf{high}_v\rangle$, this is accomplished by recursively computing the squared norm of the node's low and high edges. This subroutine visits each node at most once by virtue of dynamic programming, which stores intermedi-

Algorithm 5 Algorithms MeasurementProbability and UpdatePostMeas for respectively computing the probability of observing outcome $|0\rangle$ when measuring the top qubit of a Pauli LIMDD in the computational basis and converting the LIMDD to the post-measurement state after outcome $m \in \{0,1\}$. The subroutine SquaredNorm takes as input a Pauli LIMDD edge e, and returns $\langle e|e\rangle$. It uses a cache to store the value s of a node v.

1: **procedure** MeasurementProbability(Edge e)
2: $s_0 :=$ SquaredNorm(follow$(0, e)$)
3: $s_1 :=$ SquaredNorm(follow$(1, e)$)
4: **return** $s_0/(s_0 + s_1)$
5: **procedure** SquaredNorm(Edge $\xrightarrow{\lambda P}(v)$ **with** $\lambda \in \mathbb{C}, P \in$ Pauli$^{\mathsf{idx}(v)}$)
6: **if** $\mathsf{idx}(v) = 0$ **then return** $|\lambda|^2$
7: **if** $v \notin$ SNormCache **then** ▷ Compute result once for v and store in cache:
8: SNormCache$[v]$:= SquaredNorm(follow$(0, \xrightarrow{\mathbb{I}}(v))$) + SquaredNorm(follow$(1, \xrightarrow{\mathbb{I}}(v))$)
9: **return** $|\lambda|^2 \cdot$ SNormCache$[v]$ ▷ Retrieve result for v from cache and multiply with $|\lambda|^2$
10: **procedure** UpdatePostMeas(Edge $e \xrightarrow{\lambda P}(v)$, measurement outcome $m \in \{0,1\}$)
11: **if** $m = 0$ **then**
12: $e_r :=$ MakeEdge(follow$(0, e)$, $0 \cdot$ follow$(0, e)$)
13: **else**
14: $e_r :=$ MakeEdge$(0 \cdot$ follow$(0, e)$, follow$(1, e))$
15: **return** $1/\sqrt{\text{SquaredNorm}(e_r)} \cdot e_r$

ate results in a cache SNormCache: Node $\to \mathbb{R}$ for all recursive calls (Line 7, 8). Therefore, it runs in time $\mathcal{O}(m)$ for a diagram with m nodes.

The outcome $m \in \{0,1\}$ can then be chosen by flipping a p-biased coin. The corresponding state update is implemented by the procedure UpdatePostMeas. In order to update the state $|e\rangle = |0\rangle|e_0\rangle + |1\rangle|e_1\rangle$ after the top qubit is measured to be m, we simply construct an edge $|m\rangle|e_m\rangle$ using the MakeEdge subroutine. This state is finally normalized by multiplying (the scalar on) the resulting root edge with a normalization constant computed using squared norm.

To sample from a quantum state in the computational basis, simply repeat the measurement procedure for edge $\to(v)$ with $k = \mathsf{idx}(v)$, throw a p-biased coin to determine x_k, use follow$(x_k, \to(v))$ to go to level $k-1$ and repeat the process.

3.3.3.2 Gates with simple LIMDD algorithms

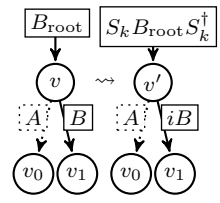
As a warm up, before we give the algorithm for arbitrary gates and Clifford gates, we first give algorithms for several gates that have a relatively simple and efficient LIMDD manipulation operation. In the case of a controlled gate, we distinguish two cases, depending whether the control or the target qubit comes first; we call these a *downward* and an *upward* controlled gate, respectively.

Here, we let L_k denote the unitary applying local gate L on qubit k, i.e., $L_k \triangleq \mathbb{I}^{\otimes n-k} \otimes L \otimes \mathbb{I}^{\otimes k-1}$.

Applying a single-qubit Pauli gate Q to qubit k of a LIMDD, by updating the diagram's root edge from A to $Q_k A$, i.e., change $A = \lambda P_n \otimes \cdots \otimes P_1$ to $\lambda P_n \otimes \cdots \otimes P_{k+1} \otimes QP_k \otimes P_{k-1} \otimes \cdots \otimes P_1$. Since only nodes —and not root edges— need be canonical, this can be done in constant time, provided that the LIMDD is stored in the natural way (uncompressed with objects and pointers).

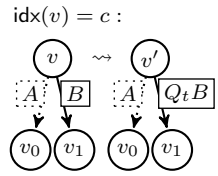
Applying any diagonal or antidiagonal single-qubit gate to the top qubit can be done efficiently, e.g., applying the **T-gate to the top qubit**. For root edge $e = \overset{B_{\text{root}}}{\longrightarrow}\!\!(v)$, we can construct $e_x = \text{FOLLOW}(x, e)$, which propagates the root edge's LIM to the root's two children. Then, for a diagonal node $\left[\begin{smallmatrix}\alpha & 0\\ 0 & \beta\end{smallmatrix}\right]$, we construct a new root node MAKEEDGE$(\alpha \cdot e_0, \beta \cdot e_1)$. For the anti-diagonal gate $\left[\begin{smallmatrix}0 & \beta\\ \alpha & 0\end{smallmatrix}\right]$, it is sufficient to note that $\left[\begin{smallmatrix}0 & \beta\\ \alpha & 0\end{smallmatrix}\right] = X \cdot \left[\begin{smallmatrix}\alpha & 0\\ 0 & \beta\end{smallmatrix}\right]$; thus, we can first apply a diagonal gate, and then an X gate, as described above.

Applying a phase gate ($S = \left[\begin{smallmatrix}1 & 0\\ 0 & i\end{smallmatrix}\right]$) to qubit with index k on $\overset{B_{\text{root}}}{\longrightarrow}\!\!(v)$ is also efficient. Algorithm 6 gives a recursive procedure. If $k < n = \mathsf{idx}(v)$ (top qubit), then note $S_k B_{\text{root}} |v\rangle = (S_k B_{\text{root}} S_k^\dagger) S_k |v\rangle$ where $S_k B_{\text{root}} S_k^\dagger$ is the new ($\mathcal{O}(n)$-computable) root $\langle$PAULI$\rangle$-LIM because S_k is a Clifford gate. Hence, we can 'push' S_k through the LIMs down the recursion, rebuilding the LIMDD in the backtrack with MAKEEDGE on Line 6 and 7. To apply S_k to v when $k = n = \mathsf{idx}(v)$, we finally multiply the high edge label with i on Line 4. Dynamic programming, using table SGATECACHE, ensures a linear amount of recursive calls in the number of nodes m. The total runtime is therefore $\mathcal{O}(mn^3)$, as MAKEEDGE's is cubic (see Section 3.4).



Algorithm 6 Apply gate S to qubit k for PAULI-LIMDD $\xrightarrow{A}(v)$. We let $n = \mathsf{idx}(v)$.

```
1: procedure SGate(Edge -A-(v) with A ∈ Pauli-LIM, k ∈ {1,...,idx(v)})
2:     if v ∉ SGateCache then                ▷ Compute result once for v and store in cache:
3:         if idx(v) = k then
4:             SGateCache[v] := MakeEdge(low_v, i · high_v)
5:         else
6:             SGateCache[v] := MakeEdge(SGate(low_v, k), SGate(high_v, k))
7:     return S_k A S_k^† · SGateCache[v]    ▷ Retrieve result from cache
```



Applying a Downward Controlled-Pauli gate CQ_t^c, where Q is a single-qubit Pauli gate, c the control qubit and t the target qubit with $t < c$, to a node v can also be done recursively. If $\mathsf{idx}(v) > c$, then since CQ_t^c is a Clifford gate, we may push it through the node's root label, and apply it to the children $\mathsf{low}(v)$ and $\mathsf{high}(v)$, similar to the S gate. Otherwise, if $\mathsf{idx}(v) = c$, then update v's high edge label as $B \mapsto Q_t B$, and do not recurse. Algorithm 7 shows the recursive procedure, which is similar to Algorithm 6 and also has $\mathcal{O}(mn^3)$ runtime.

Algorithm 7 Apply gate CX with control qubit c and target qubit t for PAULI-LIMDD $\xrightarrow{A}(v)$. We let $n = \mathsf{idx}(v)$. We can replace CX, with CY, CZ. modifying Line 4 accordingly (i.e. to Y_t, Z_t).

```
1: procedure CPauliGate(Edge -A-(v) with A ∈ Pauli-LIM, c, t with 1 ≤ c < t ≤ n)
2:     if v ∉ CPauliCache then               ▷ Compute result once for v and store in cache:
3:         if idx(v) = k then
4:             CPauliCache[v] := MakeEdge(low_v, X_t · high_v)
5:         else
6:             CPauliCache[v] := MakeEdge(CPauliGate(low_v, c, t), CPauliGate(high_v, c, t))
7:     return CX_t^c · A · CX_t^{c†} · CPauliCache[v]   ▷ Retrieve result from cache
```

Sec. 3.3.3.4 shows that all Clifford gates (including Hadamard and upward CNOT) have runtime $\mathcal{O}(n^4)$ when applied to a stabilizer state represented as a LIMDD. We first show how to apply general gates, in Sec. 3.3.3.3, as this yields some machinery required for Hadamards (specifically, a pointwise addition operation).

3.3.3.3 Applying a generic multi-qubit gate to a state

We use a standard approach [124] to represent quantum gates ($2^n \times 2^n$ unitary matrices) as LIMDDs. Here a matrix U is interpreted as a function $u(r_1, c_1, \ldots, r_n, c_n) \triangleq \langle r| U |c\rangle$ on $2n$ variables, which returns the entry of U on row r and column c. The function u is then represented using a LIMDD of $2n$ levels. The bits of r and c are interleaved to facilitate recursive descent on the structure. In particular, for $x, y \in \{0, 1\}$, the subfunction u_{xy} represents a quadrant of the matrix, namely the submatrix $u_{xy}(r_2, c_2, ..., r_n, c_n) \triangleq u(x, y, r_2, c_2, ..., r_n, c_n)$, as follows:

$$u = \overbrace{\left[\begin{matrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{matrix}\right]}^{u_{0*}} u_{*1} \tag{3.6}$$

Definition 3.4 formalizes this idea. Figure 3.7 shows a few examples of gates represented as LIMDDs.

Definition 3.4 (LIMDDs for gates)**.** A LIMDD edge $e = \overset{A}{\text{---}}\!\!\textcircled{u}$ can represent a (unitary) $2^n \times 2^n$ matrix U iff $\mathsf{idx}(u) = 2n$. The value of the matrix cell $U_{r,c}$ is defined as $\textsc{follow}(r_1 c_1 r_2 c_2 ... r_n c_n, \overset{A}{\text{---}}\!\!\textcircled{u})$ where r, c are the row and column index, respectively, with binary representation $r_1, ..., r_n$ and $c_1, ..., c_n$. The semantics of a LIMDD edge e as a matrix is denoted $[e] \triangleq U$ (as opposed to its semantics $|e\rangle$ as a vector).

The procedure ApplyGate (Algorithm 8) applies a gate U to a state $|\varphi\rangle$, represented by LIMDDs e_U and e_φ. It outputs a LIMDD edge representing $U|\varphi\rangle$. It works similar to well-known matrix-vector product algorithms for decision diagrams [124, 227], except that we also handle edge weights with LIMs (see Figure 3.8 for an illustration). Using the $\textsc{follow}(x, e)$ procedure, we write $|\varphi\rangle$ and U as

$$|\varphi\rangle = |0\rangle |\varphi_0\rangle + |1\rangle |\varphi_1\rangle \tag{3.7}$$

$$U = |0\rangle\langle 0| \otimes U_{00} + |0\rangle\langle 1| \otimes U_{01} + |1\rangle\langle 0| \otimes U_{10} + |1\rangle\langle 1| \otimes U_{11} \tag{3.8}$$

Then, on Line 6, we compute each of the four terms $U_{rc}|\varphi_c\rangle$ for row/column bits $r, c \in \{0, 1\}$. We do this by constructing four LIMDDs $f_{r,c}$ representing the states $|f_{r,c}\rangle = U_{r,c}|\varphi_c\rangle$, using four recursive calls to the ApplyGate algorithm. Next, on Line 7 and 8, the appropriate states are added, using Add (Algorithm 9), producing LIMDDs e_0 and e_1 for the states $|e_0\rangle = U_{00}|\varphi_0\rangle + U_{01}|\varphi_1\rangle$ and for $|e_1\rangle = U_{10}|\varphi_0\rangle +$

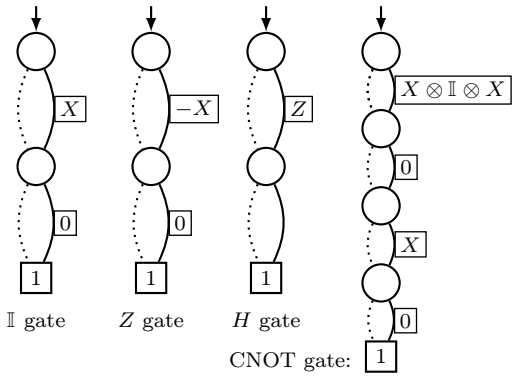


Figure 3.7: LIMDDs representing various gates.

$U_{11}|\varphi_1\rangle$. The base case of ApplyGate is the case where $n = 0$, which means U and $|v\rangle$ are simply scalars, in which case both e_U and e_φ are edges that point to the leaf.

Algorithm 8 Applies the gate $[e_U]$ to the state $|e_\varphi\rangle$. Here e_U and e_φ are LIMDD edges. The output is a LIMDD edge ψ satisfying $|\psi\rangle = [e_U]\,|e_\varphi\rangle$.

1: **procedure** ApplyGate(Edge $e_U = \xrightarrow{\lambda P}(u)$, Edge $e_\varphi = \xrightarrow{\gamma Q}(v)$ **with** idx$(u) = 2 \cdot$ idx(v))
2: **if** idx$(v) = 0$ **then return** $\xrightarrow{\lambda \cdot \gamma}\boxed{1}$ ▷ $P = Q = 1$
3: $P', Q' :=$ RootLabel($\xrightarrow{P}(u)$), RootLabel($\xrightarrow{Q}(v)$) ▷ Get canonical root labels
4: **if** $(P', u, Q', v) \notin$ Apply-cache **then** ▷ Compute result for the first time:
5: **for** $r, c \in \{0, 1\}$ **do**
6: Edge $f_{r,c} :=$ ApplyGate(follow(rc, $\xrightarrow{P'}(u)$), follow(c, $\xrightarrow{Q'}(v)$))
7: Edge $e_0 :=$ Add($f_{0,0}, f_{0,1}$)
8: Edge $e_1 :=$ Add($f_{1,0}, f_{1,1}$)
9: ApplyCache$[(P', u, Q', v)] :=$ MakeEdge(e_0, e_1) ▷ Store in cache
10: $e'_\psi :=$ Apply-cache$[(P', u, Q', v)]$ ▷ Retrieve from cache
11: **return** $\lambda\gamma \cdot e'_\psi$

Caching in ApplyGate. A straightforward way to implement dynamic programming would be to simply store all results of ApplyGate in the cache, i.e., when ApplyGate($\xrightarrow{\lambda P}(u)$, $\xrightarrow{\gamma Q}(v)$) is called, store an entry with key (P, u, Q, v) in the cache. This would allow us to retrieve the result the next time ApplyGate is called with the same parameters. However, we can do much better, in such a way that we can

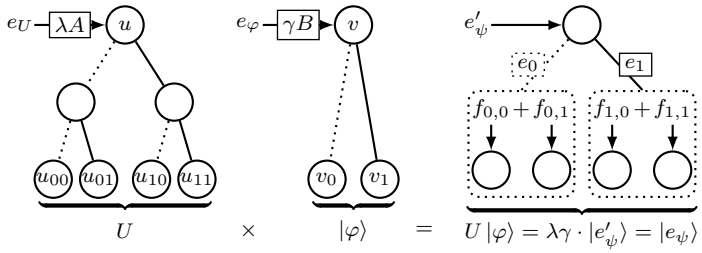


Figure 3.8: An illustration of ApplyGate (Algorithm 8), where matrix U is applied to state $B|v\rangle$, both represented as Pauli-LIMDDs. The edges $f_{0,0}$, $f_{0,1}$, etc. are the edges made on Line 6. The dotted box indicates that these states are added, using Add, producing edges e_0, e_1, which are then passed to MakeEdge, producing the result edge. For readability, not all edge labels are shown.

retrieve the result from the cache also when the procedure is called with parameters ApplyGate($\overset{A}{\text{—}}\!\!\textcircled{x}$, $\overset{B}{\text{—}}\!\!\textcircled{y}$) satisfying $[\overset{\lambda P}{\text{—}}\!\!\textcircled{u}] = [\overset{A}{\text{—}}\!\!\textcircled{x}]$ and $|\overset{\gamma Q}{\text{—}}\!\!\textcircled{v}\rangle = |\overset{B}{\text{—}}\!\!\textcircled{y}\rangle$. This can happen even when $\lambda P \neq A$ or $\gamma Q \neq B$; therefore this may prevent many recursive calls.

To this end, we store not just an edge-edge tuple from the procedure's parameters, but a *canonical* edge-edge tuple. To obtain canonical edge labels, our algorithms use the function RootLabel which returns a *canonically chosen* LIM, i.e., it holds that RootLabel($\overset{A}{\text{—}}\!\!\textcircled{v}$) = RootLabel($\overset{B}{\text{—}}\!\!\textcircled{v}$) whenever $A|v\rangle = B|v\rangle$. A specific choice for RootLabel is the lexicographic minimum of all possible root labels. In Algorithm 17, we give an $O(n^3)$-time algorithm for computing the lexicographically minimal root label, following the same strategy as the MakeEdge procedure in Sec. 3.4.2. As a last optimization, we opt to not store the scalars λ, γ in the cache (they are "factored out"), so that we can retrieve this result also when ApplyGate is called with inputs that are equal up to a complex phase. These scalars are then factored back in on Line 11 and 9.

The subroutine Add (Algorithm 9) adds two quantum states, i.e., given two LIMDDs representing $|e\rangle$ and $|f\rangle$, it returns a LIMDD representing $|e\rangle + |f\rangle$. It proceeds by simple recursive descent on the children of e and f. The base case is when both edges point to the diagram's leaf. In this case, these edges are labeled with scalars $A, B \in \mathbb{C}$, so we return the edge $\overset{A+B}{\text{—}}\!\!\textcircled{1}$.

Caching in Add. A straightforward way to implement the cache would be to store a tuple with key (A, v, B, w) in the call Add($\overset{A}{\text{—}}\!\!\textcircled{v}$, $\overset{B}{\text{—}}\!\!\textcircled{w}$). However, we can do much

Algorithm 9 Given two n-LIMDD edges e, f, constructs a new LIMDD edge a with $|a\rangle = |e\rangle + |f\rangle$.

1: **procedure** ADD(EDGE $e = \xrightarrow{A}(v)$, EDGE $f = \xrightarrow{B}(w)$ **with** $\mathsf{idx}(v) = \mathsf{idx}(w)$)
2: **if** $\mathsf{idx}(v) = 0$ **then return** $\xrightarrow{A+B}\boxed{1}$ ▷ $A, B \in \mathbb{C}$
3: **if** $v \not\approx w$ **then return** ADD($\xrightarrow{B}(w)$, $\xrightarrow{A}(v)$) ▷ Normalize for cache lookup
4: $C := \mathsf{RootLabel}(\xrightarrow{A^{-1}B}(w))$
5: **if** $(v, C, w) \notin$ ADD-CACHE **then** ▷ Compute result for the first time:
6: EDGE $a_0 :=$ ADD(FOLLOW$(0, -(v)), $FOLLOW$(0, \xrightarrow{C}(w)))$
7: EDGE $a_1 :=$ ADD(FOLLOW$(1, -(v)), $FOLLOW$(1, \xrightarrow{C}(w)))$
8: ADD-CACHE$[(v, C, w)] :=$ MAKEEDGE(a_0, a_1) ▷ Store in cache
9: **return** $A \cdot$ ADD-CACHE$[(v, C, w)]$ ▷ Retrieve from cache

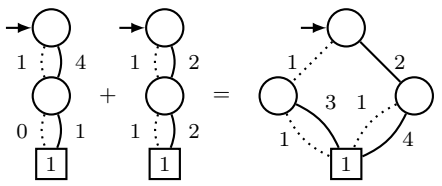


Figure 3.9: Adding two states $(0, 1, 0, 4)$ and $(1, 2, 2, 4)$ as QMDDs can cause an exponentially larger result QMDD $(1, 3, 2, 8)$ due to the loss of common factors.

better; namely, we remark that we are looking to construct the state $A|v\rangle + B|w\rangle$, and that this is equal to $A \cdot (|v\rangle + A^{-1}B|w\rangle)$. This gives us the opportunity to "factor out" the LIM A, and only store the tuple $(v, A^{-1}B, w)$. We can do even better by finding a canonically chosen LIM $C = \mathsf{RootLabel}(\xrightarrow{A^{-1}B}(w))$ (on Line 4) and storing (v, C, w) (on line Line 8). This way, we get a cache hit at Line 5 upon the call ADD($\xrightarrow{D}(v)$, $\xrightarrow{E}(w)$) whenever $A^{-1}B|w\rangle = D^{-1}E|w\rangle$. This happens of course in particular when $(A, v, B, w) = (D, v, E, w)$, but can happen in exponentially more cases; therefore, this technique works at least as well as the "straightforward" way outlined above. Finally, on Line 3, we take advantage of the fact that addition is commutative; therefore it allows us to pick a preferred order in which we store the nodes, thus improving possible cache hits by a factor two. We also use C in the recursive call at Line 6 and 7.

The worst-case runtime of ADD is $\mathcal{O}(n^3 2^n)$ (exponential as expected), where n is the number of qubits. This can happen when the resulting LIMDD is exponential in the

input sizes (bounded by 2^n), as identified for QMDDs in [113, Table 2]. The reason for this is that addition may remove any common factors, as illustrated in Figure 3.9. However, the ADD algorithm is polynomial-time when $v = w$ and v is a stabilizer state, which is sufficient to show that the Hadamard gate can be efficiently applied to stabilizers represented as LIMDD, as we demonstrate next in Sec. 3.3.3.4.

3.3.3.4 LIMDD operations for Clifford gates are polynomial time on stabilizer states

We give an algorithm for the Hadamard gate and then show that it runs in polynomial time when applied to a stabilizer state. Together with the results of Sec. 3.3.3.2, this shows that all Clifford gates can be applied to stabilizer states in polynomial time (Theorem 3.4). Indeed, since the LIMDD does not grow in size (indeed, it remains a Tower), this means all Clifford *circuits* can be simulated in polynomial time using LIMDDs.

In this section, we sketch the proof that the Hadamard gate can be applied in polynomial time; a complete proof is given in Theorem C.1 in Section C.2. The key ingredient is Lemma C.3, which shows that the ADD algorithm makes only $\mathcal{O}(n)$ many recursive calls when applied to two Pauli-equivalent stabilizer states, i.e., when it is called as ADD($\overset{A}{\text{—}}(v)$, $\overset{B}{\text{—}}(v)$).

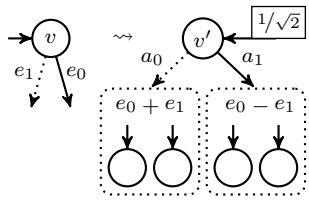
Theorem 3.4. Any Clifford gate (H, S, CNOT) can be applied in $\mathcal{O}(n^4)$ time to any (combination of) qubits to a LIMDD representing a stabilizer state.

Proof. Let $|\psi\rangle$ be an n qubit stabilizer state, represented by a LIMDD with root edge $\overset{A}{\text{—}}(v)$. By Theorem 3.1, this LIMDD is a $\langle$PAULI$\rangle$-Tower-LIMDD with $m = n$ nodes apart from the leaf.

Sec. 3.3.3.2 shows that any S-gate can be applied in time $\mathcal{O}(n^3 m)$, so we get $\mathcal{O}(n^4)$.

Theorem 3.5 shows that any Hadamard gate can be applied on any qubit in time $\mathcal{O}(n^4)$.

Sec. 3.3.3.2 shows that any downward CNOT-gate can be applied in time $\mathcal{O}(n^3 m)$, so in this case $\mathcal{O}(n^4)$. By applying Hadamard to the target and control qubits, before and after the downward CNOT, we obtain an upward CNOT, i.e., $CX_c^t = (H \otimes H)CX_t^c(H \otimes H)$, still in time $\mathcal{O}(n^4)$. □



To apply a Hadamard gate ($H = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right]$) to the first qubit, we first construct edges representing the states $|a_0\rangle = |e_0\rangle + |e_1\rangle$ and $|a_1\rangle = |e_0\rangle - |e_1\rangle$, using the ADD procedure (Algorithm 9 and multiplying the root edge with -1). Then we construct an edge representing the state $|0\rangle|a_0\rangle + |1\rangle|a_1\rangle$ using MAKEEDGE. Lastly, the complex factor on the new edge's root label is multiplied by $\frac{1}{\sqrt{2}}$. Since the Hadamard is also a Clifford gate, we can apply this operation to any qubit in the LIMDD by "pushing it through the LIMs," as we saw in Sec. 3.3.3.2. Specifically, when applying a Hadamard gate to edge $\xrightarrow{A}(v)$, we use $H_k A\,|v\rangle = (H_k A H_k^\dagger)\cdot H\,|v\rangle$, which allows us to apply a Hadamard gate to the *node* v rather than the *edge* $\xrightarrow{A}(v)$. Algorithm 10 shows the complete algorithm.

Algorithm 10 Apply gate H to qubit k for PAULI-LIMDD $\xrightarrow{A}(v)$. We let $n = \mathsf{idx}(v)$.

```
1: procedure HGate(Edge --A--(v) with A ∈ Pauli-LIM, k ∈ {1,...,idx(v)})
2:     if v ∉ HGateCache then                ▷ Compute result once for v and store in cache:
3:         if idx(v) = k then
4:             a0 := Add(low(v), high(v))
5:             a1 := Add(low(v), −high(v))
6:             HGateCache[v] := 1/√2 · MakeEdge(a0, a1)
7:         else
8:             HGateCache[v] := MakeEdge(HGate(low(v), k), HGate(high(v), k))
9:     return H_k A H_k^† · HGateCache[v]    ▷ Retrieve result from cache
```

Theorem 3.5. Let e be the root edge of an n-qubit $\langle$PAULI$\rangle$-Tower-LIMDD. Then HGATE(e, k) of Algorithm 10 takes $\mathcal{O}(n^4)$ time.

Proof sketch. A complete proof is given in Theorem C.1 in Section C.2. By virtue of the cache, HGATE is called at most once per node. Since the LIMDD is a Tower, there are only n nodes; so HGATE is called at most n times. For the node at level k, HGATE makes two calls to ADD on Line 4 and Line 5. Lemma C.3 shows that these calls to ADD each make at most $5k = \mathcal{O}(n)$ recursive calls. Each recursive call to ADD may invoke the MAKEEDGE procedure, which runs in time $\mathcal{O}(n^3)$, yielding a total worst-case running time of $\mathcal{O}(n^4)$, since $k \le n$. □

Since stabilizer states are closed under Clifford gates, one naturally expects that

⟨Pauli⟩-Tower-LIMDDs are also closed under the respective LIMDD manipulation operations. Indeed, we show this in Lemma B.2 (App. B).

3.3.4 Comparing LIMDD-based simulation with other methods

Prop. 3.1 shows exponential advantages of (Pauli-)LIMDDs over three state-of-the-art classical quantum circuit simulators: those based on QMDDs and MPS [262,346], and the Clifford + T simulator. In this section we prove the proposition, mainly using results from the current section: To show the separation between simulation with LIMDDs and Clifford + T, we present Theorem 3.6.

Our proofs often rely on the fact that LIMDDs are exponentially more succinct representations of a certain class of quantum states S that are generated by circuits with a certain (non-universal) gate set G. For instance, the stabilizer states that are generated by the Clifford gate set. LIMDD-based simulation —similar to MPS [335] and QMDD-based [374] simulation— proceeds by representing a state $|\varphi_t\rangle$ at time step t as a LIMDD φ_t. It then applies the gate $U_t \in G$ in the circuit corresponding to this time step to obtain a LIMDD φ_{t+1} with $|\varphi_{t+1}\rangle = U_t |\varphi_t\rangle$, thus yielding strong simulation at the final time step as reading amplitudes from the final LIMDD is easy (see Sec. 3.3.1).

It follows that LIMDD-based simulation is efficient provided that it can execute all gates U_t in polynomial time (in the size of the LIMDD representation), at least for the states in S. Note in particular that since the execution stays in S, i.e., $|\varphi_t\rangle \in S \implies |\varphi_{t+1}\rangle \in S$, the representation size can not grow to exponential size in multiple steps (S can be considered an inductive invariant in the style of Floyd [121] and de Bakker & Meertens [98]). On the other hand, since MPS and QMDD are exponentially sized for cluster states, they necessarily require exponential time on circuits computing this family of states.

3.3.4.1 LIMDD is exponentially faster than QMDD-based simulation

As state set S, we select the stabilizer states and for G the Clifford gates. Theorem 3.1 shows that LIMDDs for stabilizers are always quadratic in size in the number of qubits n, as the diagram contains n nodes and $n+1$ LIMs, each of size at most n (see Definition 3.3). Sec. 3.3.3.2 shows that LIMDD can execute all Clifford gates on stabilizer states in time $\mathcal{O}(n^4)$.

On the other hand, Theorem 3.2 shows that QMDDs for cluster states are exponentially sized. It follows that in simulation also, there is an exponential separation between QMDD and LIMDD, proving that $\mathrm{QSIM}_C^{\mathsf{QMDD}} = \Omega^*(2^n \cdot \mathrm{QSIM}_C^{\mathsf{LIMDD}})$ (Prop. 3.1 Item 3).

For the other direction, we now show that LIMDDs are at most a factor $\mathcal{O}(n^3)$ slower than QMDDs on any given circuit. First, a LIMDD never contains more nodes than a QMDD representing the same state (because QMDD is by definition a specialization of LIMDD, see Sec. 3.3.1). The LIMDD additionally uses $\mathcal{O}(n)$ memory per node to store two Pauli LIMs; thus, the total memory usage is at most a factor $\mathcal{O}(n)$ worse than QMDDs for any given state. The ApplyGate and Add algorithms introduced in Sec. 3.3.3.3 are very similar to the ones used for QMDDs in [124,373]. In particular, our ApplyGate and Add algorithms never make more recursive calls than those for QMDDs. However, one difference is that our MakeEdge algorithm runs in time $\mathcal{O}(n^3)$ instead of $\mathcal{O}(1)$. Therefore, in the worst case these LIMDD algorithms make the same number of recursive calls to ApplyGate and Add, in which case they are slower by a factor $\mathcal{O}(n^3)$.

Finally, Corollary 3.1 shows that the pseudo-cluster state $|\varphi\rangle$ has a polynomial representation in LIMDD. By definition of the pseudo-cluster state, post-selecting (constraining) the top qubit to 0 (or 1) yields the cluster state $|G_n\rangle$. Therefore, QMDD for the pseudo-cluster state must have exponential size, as constraining can never increase the size of DD [344, Th 2.4.1]. Together with the universal simulation discussed above, this proves that the above also holds for for a simulator based on the combination QMDD $\cup$ Stab (Prop. 3.1 Item 5).

3.3.4.2 LIMDD is exponentially faster than MPS

In Sec. 3.3.4.1, we saw that LIMDD can simulate the cluster state in polynomial time. On the other hand, Theorem 3.3 shows that MPS for cluster states are exponentially sized. It follows that in simulation also, there is an exponential separation between MPS and LIMDD, proving Prop. 3.1 Item 2.

3.3.4.3 LIMDD is exponentially faster than Clifford + T

In this section, we consider a circuit family that LIMDDs can efficiently simulate, but which is difficult for the Clifford+T simulator because the circuit contains many

T gates, assuming the Exponential Time Hypothesis (ETH, a standard complexity-theoretic assumption which is widely believed to be true). This method decomposes a given quantum circuit into a circuit consisting only of Clifford gates and the $T = \left[\begin{smallmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{smallmatrix}\right]$ gate, as explained in Section 3.2.

The circuit family, given my McClung [220], maps the input state $|0\rangle^{\otimes n}$ to the n-qubit W state $|W_n\rangle$, which is the equal superposition over computational-basis states with Hamming weight 1,

$$|W_n\rangle = \frac{1}{\sqrt{n}} \left(|100...00\rangle + |010...00\rangle + ... + |000...01\rangle\right)$$

Arunachalam et al. showed that, assuming ETH, any circuit which deterministically produces the $|W_n\rangle$ state in this way requires $\Omega(n)$ T gates [21]. Consequently, the Clifford + T simulator cannot efficiently simulate the circuit family, even when one allows for preprocessing with a compilation algorithm aiming to reduce the T-count of the circuit (such as the ones developed in [181, 313]).

Theorem 3.6 now shows that the exponential separation between simulation with LIMDD and Clifford + T, i.e., that $\text{QSIM}_C^{\text{Clifford}+T} = \Omega(2^n \cdot \text{QSIM}_C^{\text{LIMDD}})$ (Prop. 3.1 Item 1). App. D gives its proof.

Theorem 3.6. There exists a circuit family C_n such that $C_n |0\rangle^{\otimes n} = |W_n\rangle$, that Pauli-LIMDDs can efficiently simulate. Here simulation means that it constructs representations of all intermediate states, in a way which allows one to, e.g., efficiently simulate any single-qubit computational-basis measurement or compute any computational basis amplitude on any intermediate state and the output state.

We note that we could have obtained a similar result using the simpler scenario where one applies a T gate to each qubit of the $(|0\rangle + |1\rangle)^{\otimes n}$ input state. However, our goal is to show that LIMDDs can natively simulate scenarios which are relevant to quantum applications, such as the stabilizer states from the previous section. The W state is a relevant example, as several quantum communication protocols use the W state [170, 203, 206]. In contrast, the circuit with only T gates yields a product state, hence it is not relevant unless we consider it as part of a larger circuit which includes multi-qubit operations.

Lastly, it would be interesting to analytically compare LIMDD with general stabilizer rank based simulation (without assuming ETH). However, this would require finding a family of states with provably superpolynomial stabilizer rank, which is a major

open problem. Instead, we implemented a heuristic algorithm by Bravyi et al. [61] to empirically find upper bounds on the stabilizer rank and applied it to a superset of the W states, so-called Dicke states, which can be represented as polynomial-size LIMDD. The $\mathcal{O}(n^2)$-size LIMDD can be obtained via a construction by Bryant [67], since the amplitude function of a Dicke state is a symmetric function. The results hint at a possible separation but are inconclusive due to the small number of qubits which the algorithm can feasibly investigate in practice. See Section 3.6 for details.

3.4 Canonicity: Reduced LIMDDs with efficient MakeEdge algorithm

Unique representation, or canonicity, is a crucial property for the efficiency and effectiveness of decision diagrams. In the first place, it allows for circuit analysis and simplification [69, 227], by facilitating efficient manipulation operations through dynamic programming efficiently, as discussed in Sec. 3.3.3. In the second place, a reduced diagram is smaller than an unreduced diagram because it merges nodes with the same semantics. For instance, Pauli-LIMDDs allow all states in the same $\simeq_{\text{Pauli}}$ equivalence class to be merged. Here, we define a reduced Pauli-LIMDD, which is canonical.

In general, many different LIMDDs can represent a given quantum state, as illustrated in Figure 3.10. However, by imposing a small number of constraints on the diagram, listed in Definition 3.5 and visualized in Figure 3.11, we ensure that every quantum state is represented by a unique '*reduced*' Pauli-LIMDD. We present a MakeEdge algorithm (Algorithm 11 in Sec. 3.4.2) that computes a canonical node assuming its children are already canonical. The algorithms for quantum circuit simulation in Sec. 3.3.3 ensure that all intermediate LIMDDs are reduced by creating nodes exclusively through this subroutine.

3.4.1 LIMDD canonical form

The main insight used to obtain canonical decision diagrams is that a canonical form can be computed locally for each node, assuming its children are already canonical. In other words, if the diagram is constructed bottom up, starting from the leaf, it can immediately be made canonical. (This is why decision diagram manipulation

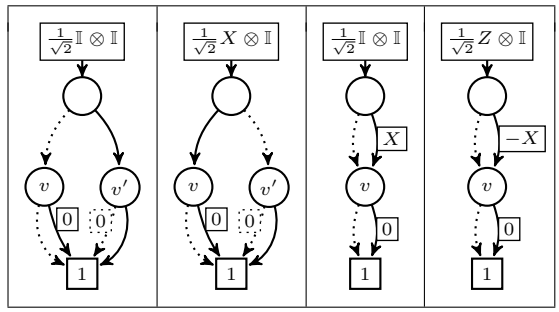


Figure 3.10: Four different Pauli-LIMDDs representing the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. From left to right: as $\mathbb{I}$-LIMDD, swapping high and low nodes v, v' by placing an X on the root LIM, merging v' into v by observing that $|v\rangle = X\,|v'\rangle$ and selecting a different high LIM $-X$ together with changing the root LIM. This section shows that selecting a unique high LIM is the most challenging, as in general many LIMs can be chosen.

algorithms always construct the diagram in the backtrack of the recursion using a typical 'MakeNode' procedure for constructing canonical nodes [124], like in Sec. 3.3.3.) For instance, a QMDD node $v \xrightarrow{\alpha} \bigcirc \xrightarrow{\beta} w$ with $\alpha, \beta \in \mathbb{C} \setminus \{0\}$ can be *reduced* into a canonical node by dividing out a common factor α and placing it on the root edge. Assuming that v, w are canonical, the resulting node $v \xrightarrow{1} \bigcirc \xrightarrow{\beta/\alpha} w$ can be stored as a tuple $(1, v, \beta/\alpha, w)$ in a hash table. Moreover, *any other node that is equal to this node up to a scalar is reduced to the same tuple with this strategy* [227] and thus merged in the hash table.

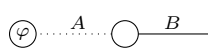
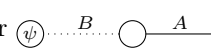
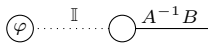
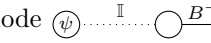
For LIMDD, we use a similar approach of dividing out 'common LIM factors.' However, we need to do additional work to obtain a unique high edge label (β/α in the example above), as the PauliLIM group is more complicated than the group of complex numbers (scalars).

Definition 3.5 gives reduction rules for LIMDDs and Figure 3.11 illustrates them. The merge (1) and low factoring (4) rules fulfill the same purpose as in the QMDD case discussed above. In a Pauli-LIMDD, we may always swap high and low edges of a node v by multiplying the root edge LIM with $X \otimes \mathbb{I}$, as illustrated in Figure 3.10. The low precedence rule (3) makes this choice deterministic, but only in case $\mathsf{low}(v) \neq \mathsf{high}(v)$. Next, the zero edges (2) rule handles the case when α ór β are zero in the above, as in principle a edge e with label 0 could point to any node on the next level k, as this always yields a 0 vector of length 2^k (see semantics below Definition 3.2). The rule forces $\mathsf{low}(v) = \mathsf{high}(v)$ in case either edge has a zero label. We explain the interaction

among the zero edges (2), low precedence (3) and low factoring (4) rules below. Finally, the high determinism rule (5) defines a deterministic function to choose LIMs on high edges, solving the most challenging problem of uniquely selecting a LIM on the high edge. We give an $\mathcal{O}(n^3)$ algorithm for this function in Sec. 3.4.2.

Definition 3.5 (Reduced LIMDD)**.** A Pauli-LIMDD is *reduced* when it satisfies the following constraints. It is *semi-reduced* if it satisfies all constraints except possibly high determinism.

1. **Merge:** No two nodes are identical: We say two nodes v, w are identical if $\mathsf{low}(v) = \mathsf{low}(w)$, $\mathsf{high}(v) = \mathsf{high}(w), \mathsf{label}(\mathsf{low}(v)) = \mathsf{label}(\mathsf{low}(w)), \mathsf{label}(\mathsf{high}(v)) = \mathsf{label}(\mathsf{high}(w))$.
2. **(Zero) edge:** For any edge $(v, w) \in \mathsf{high} \cup \mathsf{low}$, if $\mathsf{label}(v, w) = 0$, then both edges outgoing from v point to the same node, i.e., $\mathsf{high}(v) = \mathsf{low}(v) = w$.
3. **Low precedence:** Each node v has $\mathsf{low}(v) \preccurlyeq \mathsf{high}(v)$, where $\preccurlyeq$ is a total order on nodes.
4. **Low factoring:** The label on every low edge to a node v is the identity $\mathbb{I}^{\otimes \mathsf{idx}(v)}$.
5. **High determinism:** The label on the high edge of any node v is $B_{\text{high}} = \mathsf{HighLabel}(v)$, where HighLabel is a function that takes as input a semi-reduced n-Pauli-LIMDD node v, and outputs an $(n-1)$-Pauli-LIM B_{high} satisfying $|v\rangle \simeq_{\text{Pauli}} |0\rangle |\mathsf{low}(v)\rangle + |1\rangle \otimes B_{\text{high}} |\mathsf{high}(v)\rangle$. Moreover, for any other semi-reduced node w with $|v\rangle \simeq_{\text{Pauli}} |w\rangle$, it satisfies $\mathsf{HighLabel}(w) = B_{\text{high}}$. In other words, the function HighLabel is constant within an isomorphism class.

We make several observations about reduced LIMDDs. First, let us apply this definition to a state $|0\rangle \otimes A |\varphi\rangle + |1\rangle \otimes B |\psi\rangle$ with $|\varphi\rangle \not\simeq_{\text{Pauli}} |\psi\rangle$, where $A, B \in$ PauliLIM. Assume we already have canonical LIMDDs for φ and ψ (note that necessarily $\varphi \neq \psi$). We will transform this node so that it satisfies all the reduction rules above. There is a choice between representing this state as either  or  , as these are related by the isomorphism $X \otimes \mathbb{I}$. The low precedence rule resolves this choice here. Assuming $\varphi \prec \psi$, low factoring can now be realized by dividing out the LIM A, yielding a node  (with root edge $\mathbb{I} \otimes A$ as in Figure 3.11 (4)). Otherwise, if $\psi \prec \varphi$, we obtain node  with incoming edge $X \otimes B$. Finally, since there might be other LIMs B_{high} not equal to $B^{-1}A$ that yield the same state, the high determinism rule is finally needed to obtain a canonical node

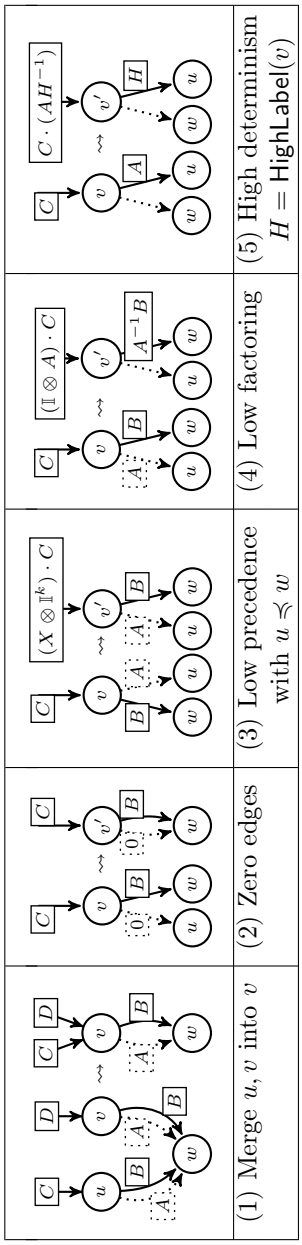
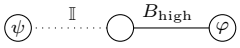
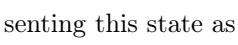
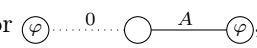
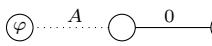


Figure 3.11: Illustrations of the reduction rules from Definition 3.5 applied at level $k+1$ (i.e., $k+1 = \mathsf{idx}(v) = \mathsf{idx}(v')$). Note that, in general, the top edges are not necessarily root edges, but could be high and low edges for nodes on level $k+2$. So, in general, there can be multiple such incoming edges (dashed and solid).

 as shown in Figure 3.12. This last step turns a semi-reduced node into a (fully) reduced node. Sec. 3.4.2 discusses it in detail.

Now, let us apply the definition to a state $|1\rangle \otimes A|\varphi\rangle$. First, notice that the zero edges rule forces $\mathsf{low}(v) = \mathsf{high}(v) = \varphi$ in this case. There is a choice between representing this state as either  or  , which denote the states $|0\rangle \otimes A|\varphi\rangle$ and $|1\rangle \otimes A|\varphi\rangle$, as these are related by the isomorphism $X \otimes \mathbb{I}$. The low factoring rule requires that the low edge label is $\mathbb{I}$, yielding a node of the form  with root label $X \otimes A$: In other words, this rule enforces swapping high and low edges, placing a X on the root label, *and* dividing out the LIM A. Consequently, the high edge must be labeled with 0, and therefore, semi-reduction, in this case, coincides with (full) reduction (no high determinism is required). Notice also that there is no reduced LIMDD for the 0-vector, because low factoring requires low edges with label $\mathbb{I}$. This is not a problem, since the 0-vector is not a quantum state.

The rules in Definition 3.5 are defined only for Pauli-LIMDDs, to which our results pertain (except for the brief mention of $\langle X\rangle$ and $\langle Z\rangle$-LIMDDs in Sec. 3.3.2). We briefly discuss alternative groups here. If G is a group without the element $X \notin G$, the reduced G-LIMDD based on the same rules is not universal (does not represent all quantum states), because the low precedence rule cannot always be satisfied, since it requires that $v_0 \preccurlyeq v_1$ for every node. Hence, in this case, reduced G-LIMDD cannot represent a state $|0\rangle|v_0\rangle + |1\rangle|v_1\rangle$ when $v_1 \prec v_0$. However, it is not difficult to formulate rules to support these groups G; for instance, when $G = \{\mathbb{I}\}$, we recover the QMDD and may use its reduction rules [374].

Nodes and edges in a reduced LIMDD need not represent normalized quantum states, just like in (unreduced) LIMDDs as explained in Sec. 3.3.1. Consider, e.g., node ℓ_2 in Figure 3.3, which represents state $[1, 1, i, i]^\top$. Because the normalization constant was divided out (see factor $1/4$ on the root edge), this state is not normalized. In fact, the root node does not need to be normalized, as even reduced LIMDDs can represent any vector (except for the zero vector).

Lastly, the literature on other decision diagrams [7,67,115] often considers a "redundant test" or "deletion" rule to remove nodes with the same high and low child. This would introduce the skipping of qubit levels, which our syntactic definition disallows, as already discussed in Footnote ‡. However, if needed Definition 3.2 could be adapted and a deletion rule could be added to Definition 3.5.

We now give a proof of Theorem 3.7, which states that reduced LIMDDs are canonical.

Theorem 3.7 (Node canonicity)**.** For each n-qubit quantum state $|\varphi\rangle$, there exists a unique reduced Pauli-LIMDD L with root node v_L such that $|v_L\rangle \simeq |\varphi\rangle$.

Proof. We use induction on the number of qubits n to show universality (the existence of an isomorphic LIMDD node) and uniqueness (canonicity).

Base case. If $n = 0$, then $|\varphi\rangle$ is a complex number λ. A reduced Pauli-LIMDD for this state is the leaf node representing the scalar 1. To show it is unique, consider that nodes v other than the leaf have an $\mathsf{idx}(v) > 0$, by the edges rule, and hence represent multi-qubit states. Since the leaf node itself is defined to be unique, the merge rule is not needed and canonicity follows.
Finally, $|\varphi\rangle$ is represented by root edge $\xrightarrow{\lambda}\boxed{1}$.

Inductive case. Suppose $n > 0$. We first show existence, and then show uniqueness.

Part 1: existence. We use the unique expansion of $|\varphi\rangle$ as $|\varphi\rangle = |0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes |\varphi_1\rangle$ where $|\varphi_0\rangle$ and $|\varphi_1\rangle$ are either $(n-1)$-qubit state vectors, or the all-zero vector. We distinguish three cases based on whether $|\varphi_0\rangle, |\varphi_1\rangle = 0$.

Case $|\varphi_0\rangle, |\varphi_1\rangle = 0$: This case is ruled out because $|\varphi\rangle \neq 0$.

Case $|\varphi_0\rangle = 0$ or $|\varphi_1\rangle = 0$: In case $|\varphi_0\rangle \neq 0$, by the induction hypothesis, there exists a Pauli-LIMDD with root node w satisfying $|w\rangle \simeq |\varphi_0\rangle$. By definition of $\simeq$, there exists an n-qubit Pauli isomorphism A such that $|\varphi_0\rangle = A|w\rangle$. We construct the following reduced Pauli-LIMDD for $|\varphi\rangle$: $(w) \overset{I}{\cdots} (v) \overset{0}{\text{———}} (w)$, adding a root edge $e_r = \overset{\mathbb{I}\otimes A}{\text{———}}(v)$ as illustrated in Figure 3.12 (left). In case $|\varphi_1\rangle \neq 0$, we do the same for root node In case $|\varphi_1\rangle \neq 0$, we do the same for root $|w\rangle \simeq |\varphi_1\rangle = A|w\rangle$, but switch the high and the low edge by instead a root edge $e_r = \overset{X\otimes A}{\text{———}}(v)$ (similar to Figure 3.11 (3)). In both cases, it is easy to check that the root node v is reduced as it can be represented by a tuple $(\mathbb{I}, w, 0, w)$, where w is canonical because of the induction hypothesis. Also in both cases, we also have $|\varphi\rangle = |e_r\rangle$ because either $|\varphi\rangle = \mathbb{I} \otimes A|v\rangle$ or $|\varphi\rangle = X \otimes A|v\rangle$.

Case $|\varphi_0\rangle, |\varphi_1\rangle \neq 0$: By applying the induction hypothesis twice, there exist Pauli-LIMDDs L and R with root nodes $|v_L\rangle \simeq |\varphi_0\rangle$ and $|v_R\rangle \simeq |\varphi_1\rangle$. The induction hypothesis implies only a 'local' reduction of LIMDDs L and R, but not automatically a reduction of their union. For instance, L might contain a node v and R a node

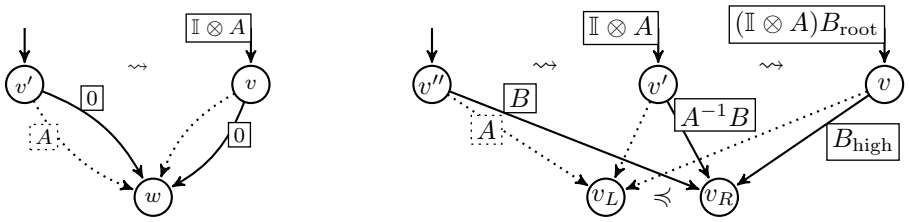


Figure 3.12: Reduced node construction in case $|\varphi_1\rangle = 0$ (left), and $|\varphi_0\rangle, |\varphi_1\rangle \neq 0$ and $v_L \preccurlyeq v_R$ (right). Not shown: for cases $|\varphi_0\rangle = 0$ and $v_R \preccurlyeq v_L$, we take instead root edge $X \otimes A$ and swap low/high edges.

w such that $v \simeq w$. While the other reduction rules ensure that v and w will be structurally the same, the induction hypothesis only applies the merge rule L and M in isolation, leaving two copies of identical nodes v, w. We can solve this by applying merge on the union of nodes in L and M, to merge any equivalent nodes, as they are already structurally equivalent by the induction hypothesis. This guarantees that (also) v_L, v_R are identical nodes.

By definition of $\simeq$, there exist n-qubit Pauli isomorphisms A and B such that $|\varphi_0\rangle = A|v_L\rangle$ and $|\varphi_1\rangle = B|v_R\rangle$. In case $v_L \preccurlyeq v_R$, we construct the following reduced Pauli-LIMDD for $|\varphi\rangle$: the root node is $(v_L) \overset{\mathbb{I}}{\cdots} (v) \overset{E}{—} (v_R)$, where E is the LIM computed by $\mathsf{HighLabel}((v_L) \overset{\mathbb{I}}{\cdots} \bigcirc \overset{A^{-1}B}{—} (v_R))$. Otherwise, if $v_R \preccurlyeq v_L$, then we construct the following reduced Pauli-LIMDD for $|\varphi\rangle$: the root node is $(v_R) \overset{I}{\cdots} (v) \overset{F}{—} (v_L)$, where $F = \mathsf{HighLabel}((v_L) \overset{\mathbb{I}}{\cdots} \bigcirc \overset{B^{-1}A}{—} (v_R))$. It is straightforward to check that, in both cases, this Pauli-LIMDD is reduced. Moreover, $|v\rangle$ isomorphic to $|\varphi\rangle$ as illustrated in Figure 3.12 (right).

Part 2: uniqueness. To show uniqueness, let L and M be reduced LIMDDs with root nodes v_L, v_M such that $|v_L\rangle \simeq |\varphi\rangle \simeq |v_M\rangle$, as follows,

$$(v_L^0) \overset{A_L}{\cdots} (v_L) \overset{B_L}{—} (v_L^1) \qquad\qquad (v_M^0) \overset{A_M}{\cdots} (v_M) \overset{B_M}{—} (v_M^1) \tag{3.9}$$

The fact that these nodes are isomorphic means that there is a Pauli isomorphism P such that $P|v_L\rangle = |v_M\rangle$. We write $P = \lambda P_{\text{top}} \otimes P_{\text{rest}} \neq 0$ where P_{top} is a single-qubit Pauli matrix and P_{rest} an $(n-1)$-qubit Pauli LIM. Expanding the semantics of v_L and

v_M, we obtain,

$$\lambda P_{\text{top}} \otimes P_{\text{rest}}(|0\rangle \otimes A_L |v_L^0\rangle + |1\rangle \otimes B_L |v_L^1\rangle) = |0\rangle \otimes A_M |v_M^0\rangle + |1\rangle \otimes B_M |v_M^1\rangle . \tag{3.10}$$

We distinguish two cases from here on: where $P_{\text{top}} \in \{\mathbb{I}, Z\}$ or $P_{\text{top}} \in \{X, Y\}$.

Case $P_{\text{top}} = I, Z$. If $P_{\text{top}} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & z \end{smallmatrix}\right]$ for $z \in \{1, -1\}$, then Equation 3.10 gives:

$$\lambda P_{\text{rest}} A_L |v_L^0\rangle = A_M |v_M^0\rangle \qquad \text{and} \qquad z\lambda P_{\text{rest}} B_L |v_L^1\rangle = B_M |v_M^1\rangle \tag{3.11}$$

By low factoring, we have $A_L = A_M = \mathbb{I}$, so we obtain $\lambda P_{\text{rest}} |v_L^0\rangle = |v_M^0\rangle$. Hence $|v_L^0\rangle$ is isomorphic with $|v_M^0\rangle$, so by the induction hypothesis, we have $v_L^0 = v_M^0$. We now show that also $v_L = v_M$ by considering two cases.

$B_L \neq 0$ and $B_M \neq 0$: then $z\lambda P_{\text{rest}} B_L |v_L^1\rangle = B_M |v_M^1\rangle$, so the nodes v_L^1 and v_M^1 represent isomorphic states, so by the induction hypothesis we have $v_L^1 = v_M^1$. We already noticed by the low factoring rule that v_L and v_M have $\mathbb{I}$ as low edge label. By the high edge rule, their high edge labels are $\mathsf{HighLabel}(v_L)$ and $\mathsf{HighLabel}(v_M)$, and since the nodes v_L and v_M are semi-reduced and $|v_L\rangle \simeq |v_M\rangle$, we have $\mathsf{HighLabel}(v_M) = \mathsf{HighLabel}(v_L)$ by definition of $\mathsf{HighLabel}$.

$B_L = 0$ or $B_M = 0$: In case $B_L = 0$, we see from Equation 3.11 that $0 = B_M |v_M^1\rangle$. Since the state vector $|v_M^1\rangle \neq 0$ by the observation that a reduced node does not represent the zero vector, it follows that $B_M = 0$. Otherwise, if $B_M = 0$, then Equation 3.11 yields $z\lambda P_{\text{rest}} B_L |v_L^1\rangle = 0$. We have $z\lambda \neq 0$, $P_{\text{rest}} \neq 0$ by definition, and we observed $|v_L^1\rangle \neq 0$ above. Therefore $B_L = 0$. In both cases, $B_L = B_M$.

We conclude that in both cases v_L and v_M have the same children and the same edge labels, so they are identical by the merge rule.

Case $P_{\text{top}} = X, Y$. If $P_{\text{top}} = \left[\begin{smallmatrix} 0 & z^* \\ z & 0 \end{smallmatrix}\right]$ for $z \in \{1, i\}$, then Equation 3.10 gives:

$$\lambda z P_{\text{rest}} A_L |v_L^0\rangle = B_M |v_M^1\rangle \qquad \text{and} \qquad \lambda z^* P_{\text{rest}} B_L |v_L^1\rangle = A_M |v_M^0\rangle .$$

By low factoring, $A_L = A_M = \mathbb{I}$, so we obtain $z\lambda P_{\text{rest}} |v_L^0\rangle = B_M |v_M^1\rangle$ and $\lambda z^* P_{\text{rest}} B_L |v_L^1\rangle = |v_M^0\rangle$. To show that $v_L = v_M$, we consider two cases.

$B_L \neq 0$ and $B_M \neq 0$: we find $|v_L^0\rangle \simeq |v_M^1\rangle$ and $|v_L^1\rangle \simeq |v_M^0\rangle$, so by the induction hypothesis, $v_L^0 = v_M^1$ and $v_L^1 = v_M^0$. By low precedence, it must be that $v_L^1 = v_M^1 = v_L^0 = v_M^0$. Now use high determinism to infer that $B_L = B_M$ as in the $P_{\text{top}} = I, Z$ case.

$B_L = 0$ or $B_M = 0$: This case leads to a contradiction and thus cannot occur. B_L cannot be zero, because then $|v_M^0\rangle$ is the all-zero vector, which we excluded. The other case: if $B_M = 0$, then it must be that $\lambda z P_{\text{rest}} A_L |v_L^0\rangle$ is zero. Since $\lambda z P_{\text{rest}} \neq 0$ and $A_L = \mathbb{I}$, it follows that $|v_L^0\rangle$ is the all-zero vector, which is again excluded.

We conclude that v_L and v_M have the same children and the same edge labels for all choices of P_{top}, so they are identical by the merge rule. □

3.4.2 The MakeEdge subroutine: Maintaining canonicity during simulation

To construct new nodes and edges, our algorithms use the MakeEdge subroutine (Algorithm 11), as discussed in Sec. 3.4.1. MakeEdge produces a reduced parent node (with root edge) given two reduced children, so that the LIMDD representation becomes canonical. Here we give the algorithm for MakeEdge and show that it runs in time $O(n^3)$ (assuming the input nodes are reduced).

The MakeEdge subroutine distinguishes two cases, depending on whether both children are non-zero vectors, which both largely follow the discussion below Definition 3.5. It works as follows:

- First it ensures low precedence, switching e_0 and e_1 if necessary at Line 3. This is also done if e_0's label A is 0 to allow for low factoring (avoiding divide by zero).
- Low factoring, i.e., dividing out the LIM A, placing it on the root node, is visualized in Figure 3.12 for the cases $e_1 = 0/e_1 \neq 0$, and done in the algorithm at Line 6,7 / 9,11.
- The zero edges rule is enforced in the $B = 0$ branch by taking $v_1 := v_0$.

- The canonical high label B_{high} is computed by GetLabels, discussed below, for the semi-reduced node $v_0 \xrightarrow{\mathbb{I}} w \xrightarrow{\hat{A}} v_1$ with $v_0 \neq v_1$. With the resulting high label, it now satisfies the high determinism rule of Definition 3.5 with $\mathsf{HighLabel}(w) = B_{\text{high}}$.
- Finally, we merge nodes by creating an entry $(v_0, B_{\text{high}}, v_1)$ in a table called the *unique table* [59] at Line 13.

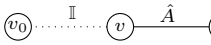
All steps except for GetLabels have complexity $O(1)$ or $O(n)$ (for checking low precedence, we use the nodes' order in the unique table). The algorithm GetLabels, which we sketch below in Sec. 3.4.2.1 and fully detail in Section 3.5, has runtime $O(n^3)$ if both input nodes are reduced, yielding an overall complexity $O(n^3)$.

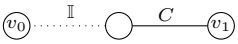
Algorithm 11 Algorithm MakeEdge takes two root edges to (already reduced) nodes v_0, v_1, the children of a new node, and returns a reduced node with root edge. It assumes that $\mathsf{idx}(v_0) = \mathsf{idx}(v_1) = n$. We indicate which lines of code are responsible for which reduction rule in Definition 3.5.

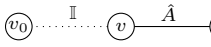
1: **procedure** MakeEdge(Edge $e_0 \xrightarrow{A} v_0$, $e_1 \xrightarrow{B} v_1$, **with** v_0, v_1 reduced, $A \neq 0$ **or** $B \neq 0$)
2: | **if** $v_0 \not\propto v_1$ **or** $A = 0$ **then** ▷ Enforce **low precedence** and enable **factoring**
3: | | **return** $(X \otimes \mathbb{I}^{\otimes n}) \cdot \text{MakeEdge}(e_1, e_0)$
4: | **if** $B = 0$ **then**
5: | | $v_1 := v_0$ ▷ Enforce **zero edges**
6: | | $v := v_0 \xrightarrow{\mathbb{I}^{\otimes n}} \bigcirc \xrightarrow{0} v_0$ ▷ Enforce **low factoring**
7: | | $B_{\text{root}} := \mathbb{I} \otimes A$ ▷ $B_{\text{root}} |v\rangle = |0\rangle \otimes A |v_0\rangle + |1\rangle \otimes B |v_1\rangle$
8: | **else**
9: | | $\hat{A} := A^{-1}B$ ▷ Enforce **low factoring**
10: | | $B_{\text{high}}, B_{\text{root}} :=$ GetLabels$(\hat{A}, v_0, v_1)$ ▷ Enforce **high determinism**
11: | | $v := v_0 \xrightarrow{\mathbb{I}^{\otimes n}} \bigcirc \xrightarrow{B_{\text{high}}} v_1$ ▷ $B_{\text{root}} |v\rangle = |0\rangle \otimes |v_0\rangle + |1\rangle \otimes A^{-1}B |v_1\rangle$
12: | | $B_{\text{root}} := (\mathbb{I} \otimes A)B_{\text{root}}$ ▷ $(\mathbb{I} \otimes A)B_{\text{root}} |v\rangle = |0\rangle \otimes A |v_0\rangle + |1\rangle \otimes B |v_1\rangle$
13: | $v_{\text{r}} :=$ Find or create unique table entry Unique$[v] = (v_0, B_{\text{high}}, v_1)$ ▷ Enforce **merge**
14: | **return** $B_{\text{root}} \xrightarrow{} v_{\text{r}}$

3.4.2.1 Choosing a canonical high-edge label

In order to choose the canonical high edge label of node v, the MakeEdge algorithm calls GetLabels (Line 10 of Algorithm 11). The function GetLabels returns a

uniquely chosen LIM B_{high} among all possible high-edge labels which yield LIMDDs representing states that are Pauli-isomorphic to $|v\rangle$. We sketch the algorithm for GetLabels here and provide the algorithm in full detail in Section 3.5 (Algorithm 12). First, we characterize the eligible high-edge labels. That is, given a semi-reduced node 

, we characterize all C such that the node 

is isomorphic to 

. Our characterization shows that, modulo some complex factor, the eligible labels C are of the form

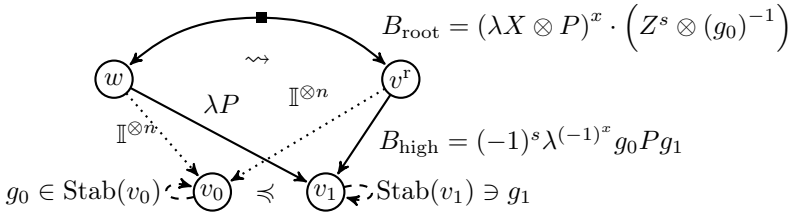
$$C \propto g_0 \cdot \hat{A} \cdot g_1, \quad \text{for } g_0 \in \text{Stab}(|v_0\rangle), g_1 \in \text{Stab}(|v_1\rangle) \tag{3.12}$$

where $\text{Stab}(|v_0\rangle)$ and $\text{Stab}(|v_1\rangle)$ are the stabilizer subgroups of $|v_0\rangle$ and $|v_1\rangle$, i.e., the already reduced children of our input node v. Note that the set of eligible high-edge labels might be exponentially large in the number of qubits. Fortunately, eq. (3.12) shows that this set has a polynomial-size description by storing only the generators of the stabilizer subgroups.

Our algorithm chooses the lexicographically smallest eligible label, i.e., the smallest C of the form $C \propto g_0 \hat{A} g_1$ (the definition of 'lexicographically smallest' is given in Sec. 3.5.2). To this end, we use two subroutines: (1) an algorithm which finds (a generating set of) the stabilizer group $\text{Stab}(|v\rangle)$ of a LIMDD node v; and (2) an algorithm that uses these stabilizer subgroups of the children nodes to choose a unique representative of the eligible-high-label set from eq. (3.12).

For (1), we use an algorithm which recurses on the children nodes. First, we note that, if the Pauli LIM A stabilizes both children, then $\mathbb{I} \otimes A$ stabilizes the parent node. Therefore, we compute (a generating set for) the intersection of the children's stabilizer groups. Second, our method finds out whether the parent node has stabilizers of the form $P_n \otimes A$ for $P_n \in \{X, Y, Z\}$. This requires us to decide whether certain cosets of the children's stabilizer groups are empty. These groups are relatively simple, since, modulo phase, they are isomorphic to a binary vector space, and cosets are hyperplanes. We can therefore rely in large part on existing algorithms for linear algebra in vector spaces. The difficult part lies in dealing with the non-abelian aspects of the Pauli group. We provide the full algorithm, which is efficient, also in Section 3.5.

Our algorithm for (2) applies a variant of Gauss-Jordan elimination to the generating sets of $\text{Stab}(|v_0\rangle)$ and $\text{Stab}(|v_1\rangle)$ to choose g_0 and g_1 in eq. (3.12) which, when multiplied with $\hat{A}$ as in eq. (3.12), yield the smallest possible high label C. (We recall that Gauss-Jordan elimination, a standard linear-algebra technique, is applicable here



Choose $s, x \in \{0,1\}, g_0 \in \text{Stab}(v_0), g_1 \in \text{Stab}(v_1)$ s.t. B_{high} is minimal and $x = 0$ if $v_0 \neq v_1$.

Figure 3.13: Illustration of finding a canonical high label for a semi-reduced node w, yielding a reduced node v^{r}. The chosen high label is the minimal element from the set of eligible high labels based on stabilizers g_0, g_1 of v_0, v_1 (drawn as self loops). The minimal element holds a factor $\lambda^{(-1)^x}$ for some $x \in \{0,1\}$. There are two cases: if $v_0 \neq v_1$ or $x = 0$, then the factor is λ and the root edge should be adjusted with an $\mathbb{I}$ or Z on the root qubit. The other case, $x = 1$, leads to an additional multiplication with an X on the root qubit.

because the stabilizer groups are group isomorphic to binary vector spaces, see also Sec. 3.5.2). We explain the full algorithm in Section 3.5.

3.4.2.2 Checking whether two LIMDDs are Pauli-equivalent

To check whether two states represented as LIMDDs are Pauli-equivalent, it suffices to check whether they have the same root node. Namely, due to canonicity, and in particular the Merge rule (in Definition 3.5), there is a unique LIMDD representing a quantum state up to phase and local Pauli operators.

3.5 Efficient algorithms for choosing a canonical high label

Here, we present an efficient algorithm which, on input Pauli-LIMDD node $v_0 \overset{\mathbb{I}}{\cdots} w \overset{\lambda P}{—} v_1$, returns a canonical choice for the high label B_{high} (algorithm GetLabels, in Algorithm 12). By *canonical*, we mean that it returns the same high label for any two nodes in the same isomorphism equivalence class, i.e., for any two nodes v, w for which $|v\rangle \simeq_{\text{Pauli}} |w\rangle$.

This section is structured as follows. In Sec. 3.5.1, we identify the canonical high label

that we are after – the minimum element of a certain set – and presents an algorithm GetLabels, which finds it. The algorithm GetLabels calls several subroutines, which are presented in Sec. 3.5.2, Sec. 3.5.3 and Sec. 3.5.4. present algorithms which find this canonical label. Sec. 3.5.2 presents algorithms operating on group of Pauli operators.

3.5.1 Choosing a canonical high label

We first characterize all eligible labels B_{high} in terms of the stabilizer subgroups of the children nodes v_0, v_1, denoted as $\text{Stab}(v_0)$ and $\text{Stab}(v_1)$ (see Section 3.2 for the definition of stabilizer subgroup). Then, we provide the algorithm GetLabels which correctly finds the lexicographically minimal eligible label (and corresponding root label), and runs in time $O(n^3)$ where n is the number of qubits.

Figure 3.13 illustrates this process. In the figure, the left node w summarizes the status of the MakeEdge algorithm on Line 10, when this algorithm has enough information to construct the semi-reduced node $v_0 \xrightarrow{\mathbb{I}^{\otimes n}} w \xrightarrow{\lambda P} v_1$, shown on the left. The node v^r, on the right, is the canonical node, and is obtained by replacing w's high edge's label by the canonical label B_{high}. This label is chosen by minimizing the expression $B_{\text{high}} = (-1)^s \lambda^{(-1)^x} g_0 P g_1$, where the minimization is over $s, x \in \{0,1\}, g_0 \in Stab(|v_0\rangle), g_1 \in Stab(|v_1\rangle)$, subject to the constraint that $x = 0$ if $v_0 \neq v_1$. We have $|w\rangle \simeq_{\text{Pauli}} |v^r\rangle$ by construction as intended, namely, they are related via $|w\rangle = B_{\text{root}} |v^r\rangle$. Theorem 3.8 shows that this way to choose the high label indeed captures all eligible high labels, i.e., a node $v_0 \xrightarrow{\mathbb{I}} v^r \xrightarrow{B_{\text{high}}} v_1$ is isomorphic to $|w\rangle$ if and only if B_{high} is of this form.

Theorem 3.8 (Eligible high-edge labels)**.** Let $v_0 \xrightarrow{\mathbb{I}^{\otimes n}} w \xrightarrow{\lambda P} v_1$ be a semi-reduced n-qubit node in a Pauli-LIMDD, where v_0, v_1 are reduced, P is a Pauli string and $\lambda \neq 0$. For all nodes $v = v_0 \xrightarrow{\mathbb{I}^{\otimes n}} v \xrightarrow{B_{\text{high}}} v_1$, it holds that $|w\rangle \simeq |v\rangle$ if and only if

$$B_{\text{high}} = (-1)^s \cdot \lambda^{(-1)^x} g_0 P g_1 \tag{3.13}$$

for some $g_0 \in \text{Stab}(v_0), g_1 \in \text{Stab}(v_1), s, x \in \{0,1\}$ and $x = 0$ if $v_0 \neq v_1$. An isomorphism mapping $|w\rangle$ to $|v\rangle$ is

$$B_{\text{root}} = (X \otimes \lambda P)^x \cdot (Z^s \otimes (g_0)^{-1}). \tag{3.14}$$

Proof. It is straightforward to verify that the isomorphism B_{root} in eq. (3.14) indeed

maps $|w\rangle$ to $|v\rangle$ (as $x = 1$ implies $v_0 = v_1$), which shows that $|w\rangle \simeq |v\rangle$. For the converse direction, suppose there exists an n-qubit Pauli LIM C such that $C|w\rangle = |v\rangle$, i.e.,

$$C\left(|0\rangle \otimes |v_0\rangle + \lambda |1\rangle \otimes P|v_1\rangle\right) = |0\rangle \otimes |v_0\rangle + |1\rangle \otimes B_{\text{high}}|v_1\rangle\,. \tag{3.15}$$

We show that if B_{high} satisfies eq. (3.15), then it has a decomposition as in eq. (3.13). We write $C = C_{\text{top}} \otimes C_{\text{rest}}$ where C_{top} is a single-qubit Pauli operator and C_{rest} is an $(n-1)$-qubit Pauli LIM (or a complex number $\neq 0$ if $n = 1$). We treat the two cases $C_{\text{top}} \in \{\mathbb{I}, Z\}$ and $C_{\text{top}} \in \{X, Y\}$ separately:

(a) **Case** $C_{\text{top}} \in \{\mathbb{I}, Z\}$. Then $C_{\text{top}} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & (-1)^y \end{smallmatrix}\right]$ for $y \in \{0, 1\}$. In this case, Equation 3.15 implies $C_{\text{top}}|0\rangle\, C_{\text{rest}}|v_0\rangle = |0\rangle|v_0\rangle$, so $C_{\text{rest}}|v_0\rangle = |v_0\rangle$, in other words $C_{\text{rest}} \in \text{Stab}(|v_0\rangle)$. Moreover, Equation 3.15 implies $(-1)^y \lambda C_{\text{rest}} P|v_1\rangle = B_{\text{high}}|v_1\rangle$, or, equivalently, $(-1)^{-y}\lambda^{-1}P^{-1}C_{\text{rest}}^{-1}B_{\text{high}} \in \text{Stab}(v_1)$. Hence, by choosing $s = y$ and $x = 0$, we compute

$$(-1)^y \lambda^{(-1)^0} \underbrace{C_{\text{rest}}}_{\in \text{Stab}(v_0)} P \underbrace{(-1)^{-y}\lambda^{-1}P^{-1}C_{\text{rest}}^{-1}B_{\text{high}}}_{\in \text{Stab}(v_1)} = \frac{(-1)^y \lambda^{(-1)^0}}{(-1)^y \lambda} B_{\text{high}} = B_{\text{high}}$$

(b) **Case** $C_{\text{top}} \in \{X, Y\}$. Write $C_{\text{top}} = \left[\begin{smallmatrix} 0 & z^{-1} \\ z & 0 \end{smallmatrix}\right]$ where $z \in \{1, i\}$. Now, eq. (3.15) implies

$$zC_{\text{rest}}|v_0\rangle = B_{\text{high}}|v_1\rangle \qquad \text{and} \qquad z^{-1}\lambda C_{\text{rest}} P|v_1\rangle = |v_0\rangle\,. \tag{3.16}$$

From Equation 3.16, we first note that $|v_0\rangle$ and $|v_1\rangle$ are isomorphic, so by Corollary 3.7, and because the diagram has merged these two nodes, we have $v_0 = v_1$. Consequently, we find from Equation 3.16 that $z^{-1}C_{\text{rest}}^{-1}B_{\text{high}} \in \text{Stab}(v_0)$ and $z^{-1}\lambda C_{\text{rest}}P \in \text{Stab}(v_1)$. Now choose $x = 1$ and choose s such that $(-1)^s \cdot z^{-2}C_{\text{rest}}^{-1}B_{\text{high}}C_{\text{rest}} = B_{\text{high}}$ (recall that Pauli LIMs either commute or anticommute, so $B_{\text{high}}C_{\text{rest}} = \pm C_{\text{rest}}B_{\text{high}}$). This yields:

$$(-1)^s\lambda^{-1}\cdot\underbrace{z^{-1}C_{\text{rest}}^{-1}B_{\text{high}}}_{\in \text{Stab}(v_0)}\cdot P\cdot\underbrace{z^{-1}\lambda P C_{\text{rest}}}_{\in \text{Stab}(v_1)} = \lambda^{-1}\cdot\lambda\cdot(-1)^s z^{-2}\cdot\left(C_{\text{rest}}^{-1}B_{\text{high}}C_{\text{rest}}\right) = B_{\text{high}}$$

where we used the fact that $P^2 = \mathbb{I}^{\otimes(n-1)}$ because P is a Pauli string.

□

Corollary 3.2. As a corollary of Theorem 3.8, we find that taking, as in Figure 3.13,

$$\mathsf{HighLabel}(v_0 \overset{\mathbb{I}}{\cdots} v \overset{\lambda P}{—} v_1) = \min_{i,s,x\in\{0,1\},g_i\in\mathrm{Stab}(v_i)} \left(\left\{(-1)^s \cdot \lambda^{(-1)^x} \cdot g_0 \cdot P \cdot g_1 \mid x \neq 1 \text{ if } v_0 \neq v_1\right\}\right)$$

yields a proper implementation of HighLabel as required by Definition 3.5, because it considers all possible B_{high} such that $|v\rangle \simeq_{\text{PAULI}} |0\rangle|v_0\rangle + |1\rangle \otimes B_{\text{high}}|v_1\rangle$.

A naive implementation for GetLabels would follow the possible decompositions of eligible LIMs (see Equation 3.13) and attempt to make this LIM smaller by greedy multiplication, first with stabilizers of $g_0 \in \mathrm{Stab}(v_0)$, and then with stabilizers $g_1 \in \mathrm{Stab}(v_1)$. To see why this does not work, consider the following example: the high edge label is Z and the stabilizer subgroups are $\mathrm{Stab}(v_0) = \langle X\rangle$ and $\mathrm{Stab}(v_1) = \langle Y\rangle$. Then the naive algorithm would terminate and return Z because $X, Y > Z$, which is incorrect since the high-edge label $X \cdot Z \cdot Y = -i\mathbb{I}$ is smaller than Z.

Algorithm 12 Algorithm for finding LIMs B_{high} and B_{root} required by MakeEdge. Its parameters represent a semi-reduced node $v_0 \overset{\mathbb{I}}{\cdots} v \overset{\lambda P}{—} v_1$ and it returns LIMs $B_{\text{high}}, B_{\text{root}}$ such that $|v\rangle = B_{\text{root}}|w\rangle$ with $v_0 \overset{\mathbb{I}}{\cdots} w \overset{B_{\text{high}}}{—} v_1$. The LIM B_{high} is chosen canonically as the lexicographically smallest from the set characterized in Theorem 3.8. It runs in $O(n^3)$-time (with n the number of qubits), provided `GetStabilizerGenSet` has been computed for children v_0, v_1 (an amortized cost).

1: **procedure** GetLabels(PauliLim λP, Node v_0, v_1 **with** $\lambda \neq 0$ and v_0, v_1 reduced)
   Output: canonical high label B_{high} and root label B_{root}
2: $G_0, G_1 :=$ `GetStabilizerGenSet`(v_0), `GetStabilizerGenSet`(v_1)
3: $(g_0, g_1) :=$ ArgLexMin$(G_0, G_1, \lambda P)$
4: **if** $v_0 = v_1$ **then**
5: $\quad (x, s) := \underset{(x,s)\in\{0,1\}^2}{\arg\min} \left\{(-1)^s \lambda^{(-1)^x} g_0 P g_1\right\}$
6: **else**
7: $\quad x := 0$
8: $\quad s := \underset{s\in\{0,1\}}{\arg\min} \{(-1)^s \lambda g_0 P g_1\}$
9: $B_{\text{high}} := (-1)^s \cdot \lambda^{(-1)^x} \cdot g_0 \cdot P \cdot g_1$
10: $B_{\text{root}} := (X \otimes \lambda P)^x \cdot (Z^s \otimes (g_0)^{-1})$
11: **return** $(B_{\text{high}}, B_{\text{root}})$

To overcome this, we consider the group closure of *both* $\mathrm{Stab}(v_0)$ *and* $\mathrm{Stab}(v_1)$. See Algorithm 12 for the $O(n^3)$-algorithm for GetLabels, which proceeds in two steps. In the first step (Line 3), we use the subroutine ArgLexMin for finding the minimal

Pauli LIM A such that $A = \lambda P \cdot g_0 \cdot g_1$ for $g_0 \in \mathrm{Stab}(v_0), g_1 \in \mathrm{Stab}(v_1)$. We will explain and prove correctness of this subroutine below in Sec. 3.5.4. In the second step (Line 4-8), we follow Corollary 3.2 by also minimizing over x and s. Finally, the algorithm returns B_{high}, the minimum of all eligible edge labels according to Corollary 3.2, together with a root edge label B_{root} which ensures the represented quantum state remains the same.

Below, we will explain $O(n^3)$-time algorithms for finding generating sets for the stabilizer subgroup of a reduced node and for ArgLexMin. Since all other lines in Algorithm 12 can be performed in linear time, its overall runtime is $O(n^3)$.

3.5.2 Efficient linear-algebra algorithms for stabilizer subgroups

In this section, we present preliminaries that are used in algorithms which ensure LIMDD canonicity, presented in Sec. 3.5.1.

In Section 3.2, we defined the stabilizer group for an n-qubit state $|\varphi\rangle$ as the group of Pauli operators $A \in \mathrm{Pauli}_n$ which stabilize $|\varphi\rangle$, i.e. $A|\varphi\rangle = |\varphi\rangle$. Here, we explain existing efficient algorithms for solving various tasks regarding stabilizer groups (whose elements commute with each other). We also outline how the algorithms can be extended and altered to work for general PauliLIMs, which do not necessarily commute. For sake of clarity, in the explanation below we first ignore the scalar λ of a PauliLIM or Pauli element λP. At the end, we explain how the scalars can be taken into account when we use these algorithms as subroutine in LIMDD operations.

Any n-qubit Pauli string can (modulo factor $\in \{\pm 1, \pm i\}$) be written as $(X^{x_n} Z^{z_n}) \otimes ... \otimes (X^{x_1} Z^{z_1})$ for bits $x_j, z_j, 1 \leq j \leq n$. We can therefore write an n-qubit Pauli string P as a length-$2n$ binary vector as follows [3],

$$(\underbrace{x_n, x_{n-1}, ...x_1}_{\text{X block}} | \underbrace{z_n, z_{n-1}, ..., z_1}_{\text{Z block}}),$$

where we added the horizontal bar (|) only to guide the eye. We will refer to such vectors as *check vectors*. For example, we have $X \sim (1,0)$ and $Z \otimes Y \sim (0,1|1,1)$. This equivalence induces an ordering on Pauli strings following the lexicographic ordering on bit strings. For example, $X < Y$ because $(1|0) < (1|1)$ and $Z \otimes \mathbb{I} < Z \otimes X$ because $(00|10) < (01|10)$.

A set of k Pauli strings thus can be written as $2n \times k$ binary matrix, often called *check matrix*, as the following example shows.

$$\begin{pmatrix} X & \otimes & X & \otimes & X \\ \mathbb{I} & \otimes & Z & \otimes & Y \end{pmatrix} \sim \left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right).$$

Furthermore, if P, Q are Pauli strings corresponding to binary vectors $(\vec{x}^P, \vec{z}^P)$ and $(\vec{x}^Q, \vec{z}^Q)$, then

$$P \cdot Q \propto \bigotimes_{j=1}^{n} \left(X^{x_j^P} Z^{z_j^P}\right)\left(X^{x_j^Q} Z^{z_j^Q}\right) = \bigotimes_{j=1}^{n} \left(X^{x_j^P \oplus x_j^Q} Z^{z_j^P \oplus z_j^Q}\right)$$

and therefore the group of n-qubit Pauli strings with multiplication (disregarding factors) is group isomorphic to the vector space $\{0,1\}^{2n}$ (i.e., $\mathbb{F}_2^{2n}$) with bitwise addition $\oplus$ (i.e., exclusive or; 'xor'). Consequently, many efficient algorithms for linear-algebra problems carry over to sets of Pauli strings. In particular, if $G = \{g_1, ..., g_k\}$ are length$-2n$ binary vectors (/ n-qubit Pauli strings) with $k \leq n$, then we can efficiently perform the following operations.

RREF: bring G into a reduced-row echelon form (RREF) using Gauss-Jordan elimination (both are standard linear algebra notions) where each row in the check matrix has strictly more leading zeroes than the row above. The RREF is achievable by $O(k^2)$ row additions (/ multiplications modulo factor) and thus $O(k^2 \cdot n)$ time (see [321] for a similar algorithm). In the RREF, the first 1 after the leading zeroes in a row is called a 'pivot'.

Construct Minimal-size Generator Set convert G to a (potentially smaller) set G' by performing the RREF procedure and discarding resulting all-zero rows. It holds that $\langle G \rangle = \langle G' \rangle$, i.e., these sets generate the same group modulo phase.

Membership: determining whether a given a vector (/ Pauli string) h has a decomposition in elements of G. This can be done by obtaining minimal-size generating sets H_1, H_2 for the sets G and $G \cup \{h\}$, respectively. Then the generating sets have the same number of elements (i.e., rows) if and only if $h \in \langle G \rangle$; otherwise, if $h \notin \langle G \rangle$, it holds that $|H_2| = |H_1| + 1$.

Intersection: determine all Pauli strings which, modulo a factor, are contained in both G_A and G_B, where G_A, G_B are generator sets for n-qubit stabilizer subgroups. More specifically, we obtain the generator set of this group, i.e., we obtain a set

G_C such that $\langle G_C\rangle = \langle G_A\rangle \cap \langle G_B\rangle$. This can be achieved using the Zassenhaus algorithm [212] for computing the intersection of two subspaces of a vector space, in time $O(n^3)$.

Division remainder: given a vector h (/ Pauli string h), determine $h^{\text{rem}} := \min_{g\in\langle G\rangle}\{g \oplus h\}$ (minimum in the lexicographic ordering). We do so in the check matrix picture by bringing G into RREF, and then making the check vector of h contain as many zeroes as possible by adding rows from G:

1: **for** column index $j = 1$ to $2n$ **do**
2: | **if** $h_j = 1$ and G has a row g_i with its pivot at position j **then** $h := h \oplus g_i$

The resulting h is h^{rem}. This algorithm's runtime is dominated by the RREF step; $O(n^3)$.

To include the scalar into the representation, we remark that Pauli LIMs that appear as labels on diagrams may have $\lambda \in \mathbb{C}$, i.e., any complex number is allowed. Therefore, to store LIMs, we use a minor extension to the check vector form introduced above, in order to also include the phase. Specifically, the phase is stored using two real numbers, by writing $\lambda = r \cdot e^{i\theta}$ with $r \in \mathbb{R}_{>0}$ and $\theta \in [0, 2\pi)$. Consequently, the check vector has $2n + 2$ entries, where the last entries store r and θ, e.g.:

$$\begin{pmatrix} 3X & \otimes & X & \otimes & X \\ -\frac{1}{2}i\mathbb{I} & \otimes & Z & \otimes & Y \end{pmatrix} \sim \left(\begin{array}{ccc|ccc|cc} 1 & 1 & 1 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & \frac{1}{2} & \frac{3\pi}{2} \end{array}\right)$$

where we used $3 = 3 \cdot e^{i\cdot 0}$ and $-\frac{1}{2}i = \frac{1}{2} \cdot e^{3\pi i/2}$. This extended check vector also easily allows a total ordering, namely, we simply use the ordering on real numbers for r and θ. For example, $(1, 1, |0, 0|2, \frac{1}{2}) < (1, 1|1, 0|3, 0)$. Let us stress that the factor encoding (r, θ) is less significant than the Pauli string encoding $(x_n, ..., x_1|z_n, ..., z_1)$. As a consequence, we can greedily determine the minimum of two Pauli operators, by reading their check vectors from left to right.

Finally, we emphasize that the algorithms above rely on bitwise xor-ing, which is a commutative operation. Since conventional (i.e., factor-respecting) multiplication of Pauli operators is not commutative, the algorithms above are not straightforwardly applicable to arbitrary PauliLIM$_n$ input. (When the input consist of pairwise commuting Pauli operators, such as stabilizer subgroups [3], the algorithms can be made to work by adjusting row addition to keep track of the scalar.) Fortunately, since Pauli strings either commute or anti-commute, row addition may only yield factors up to

the ± sign, not the resulting Pauli strings. This feature, combined with the stipulated order assigning least significance to the factor, enables us to invoke the algorithms above as subroutine. We do so in Sec. 3.4.2.1 and Sec. 3.5.3.

3.5.3 Constructing the stabilizer subgroup of a LIMDD node

In this section, we give a recursive subroutine `GetStabilizerGenSet` to construct the stabilizer subgroup $\mathrm{Stab}(|v\rangle) = \{A \in \textsc{PauliLIM}_n \mid A|v\rangle = |v\rangle\}$ of an n-qubit LIMDD node v (see Section 3.2). This subroutine is used by the algorithm GetLabels to select a canonical label for the high edge and root edge. If the stabilizer subgroup of v's children have been computed already, `GetStabilizerGenSet`'s runtime is $O(n^3)$. `GetStabilizerGenSet` returns a generating set for the group $\mathrm{Stab}(|v\rangle)$. Since these stabilizer subgroups are generally exponentially large in the number of qubits n, but they have at most n generators, storing only the generators instead of all elements may save an exponential amount of space. Because any generator set G of size $|G| > n$ can be brought back to at most n generators in time $\mathcal{O}(|G| \cdot n^2)$ (see Sec. 3.5.2), we will in the derivation below show how to obtain generator sets of size linear in n and leave the size reduction implicit. We will also use the notation $A \cdot G$ and $G \cdot A$ to denote the sets $\{A \cdot g \mid g \in G\}$ and $\{g \cdot A \mid g \in G\}$, respectively, when A is a Pauli LIM.

We now sketch the derivation of the algorithm. The base case of the algorithm is the Leaf node of the LIMDD, representing the number 1, which has stabilizer group $\{1\}$. For the recursive case, we wish to compute the stabilizer group of a reduced n-qubit node $v = \overset{v_0}{\bigcirc} \cdots \overset{\mathbb{I}}{} \cdots \overset{v}{\bigcirc} \overset{B_{\text{high}}}{\text{———}} \overset{v_1}{\bigcirc}$. If $B_{\text{high}} = 0$, then it is straightforward to see that $\lambda P_n \otimes P' |v\rangle = |v\rangle$ implies $P_n \in \{\mathbb{I}, Z\}$, and further that $\mathrm{Stab}(|v\rangle) = \langle\{P_n \otimes g \mid g \in G_0, P_n \in \{\mathbb{I}, Z\}\}\rangle$, where G_0 is a stabilizer generator set for v_0.

Otherwise, if $B_{\text{high}} \neq 0$, then we expand the stabilizer equation $\lambda P |v\rangle = |v\rangle$:

$$\lambda P_n \otimes P' \left(|0\rangle \otimes |v_0\rangle + |1\rangle \otimes B_{\text{high}} |v_1\rangle\right) = |0\rangle \otimes |v_0\rangle + |1\rangle \otimes B_{\text{high}} |v_1\rangle \text{, which implies:}$$

$$\lambda P' \left|v_0\right\rangle = \left|v_0\right\rangle \text{ and } z\lambda P' B_{\text{high}} \left|v_1\right\rangle = B_{\text{high}} \left|v_1\right\rangle \quad \textbf{if } P_n = \left[\begin{smallmatrix} 1 & 0 \\ 0 & z \end{smallmatrix}\right] \text{ with } z \in \{1, -1\} \tag{3.17}$$

$$y^* \lambda P' B_{\text{high}} \left|v_1\right\rangle = \left|v_0\right\rangle \text{and } \lambda P' \left|v_0\right\rangle = y^* B_{\text{high}} \left|v_1\right\rangle \qquad \textbf{if } P_n = \begin{bmatrix} 0 & y^* \\ y & 0 \end{bmatrix}, \text{ with } y \in \{1, i\} \tag{3.18}$$

The stabilizers can therefore be computed according to Equation 3.17 and 3.18 as follows.

$$\begin{aligned} \text{Stab}(\left|v\right\rangle) = \bigcup_{z=\in\{1,-1\}, y\in\{1,i\}} & \left[\begin{smallmatrix} 1 & 0 \\ 0 & z \end{smallmatrix}\right] \otimes \left(\text{Stab}(\left|v_0\right\rangle) \cap z \cdot \text{Stab}(B_{\text{high}} \left|v_1\right\rangle)\right) \\ & \cup \begin{bmatrix} 0 & y \\ y^* & 0 \end{bmatrix} \otimes \left(\text{Iso}(y^* B_{\text{high}} \left|v_1\right\rangle, \left|v_0\right\rangle) \cap \text{Iso}(\left|v_0\right\rangle, y^* B_{\text{high}} \left|v_1\right\rangle)\right) \end{aligned} \tag{3.19}$$

where $\text{Iso}(v, w)$ denotes the set of Pauli isomorphisms A which map $\left|v\right\rangle$ to $\left|w\right\rangle$ and we have denoted $\pi \cdot G := \{\pi \cdot g \mid g \in G\}$ for a set G and a single operator π. Lemma 3.1 shows that such an isomorphism set can be expressed in terms of the stabilizer group of $\left|v\right\rangle$.

Lemma 3.1. Let $\left|\varphi\right\rangle$ and $\left|\psi\right\rangle$ be quantum states on the same number of qubits. Let π be a Pauli isomorphism mapping $\left|\varphi\right\rangle$ to $\left|\psi\right\rangle$. Then the set of Pauli isomorphisms mapping $\left|\varphi\right\rangle$ to $\left|\psi\right\rangle$ is $\text{Iso}(\left|v\right\rangle, \left|w\right\rangle) = \pi \cdot \text{Stab}(\left|\varphi\right\rangle)$. That is, the set of isomorphisms $\left|\varphi\right\rangle \to \left|\psi\right\rangle$ is a coset of the stabilizer subgroup of $\left|\varphi\right\rangle$.

Proof. If $P \in \text{Stab}(\left|\varphi\right\rangle)$, then $\pi \cdot P$ is an isomorphism since $\pi \cdot P \left|\varphi\right\rangle = \pi \left|\varphi\right\rangle = \left|\psi\right\rangle$. Conversely, if σ is a Pauli isomorphism which maps $\left|\varphi\right\rangle$ to $\left|\psi\right\rangle$, then $\pi^{-1}\sigma \in \text{Stab}(\left|\varphi\right\rangle)$ because $\pi^{-1}\sigma \left|\varphi\right\rangle = \pi^{-1} \left|\psi\right\rangle = \left|\varphi\right\rangle$. Therefore $\sigma = \pi(\pi^{-1}\sigma) \in \pi \cdot \text{Stab}(\left|\varphi\right\rangle)$. □

With Lemma 3.1 we can rewrite eq. (3.19) as

$$\begin{aligned} \text{Stab}(\left|v\right\rangle) = & \mathbb{I} \otimes \underbrace{\left(\text{Stab}(\left|v_0\right\rangle) \cap \text{Stab}(B_{\text{high}} \left|v_1\right\rangle)\right)}_{\text{stabilizer subgroup}} \\ & \cup Z \otimes \underbrace{\left(\mathbb{I} \cdot \text{Stab}(\left|v_0\right\rangle) \cap -\mathbb{I} \cdot \text{Stab}(B_{\text{high}} \left|v_1\right\rangle)\right)}_{\text{isomorphism set}} \\ & \cup \bigcup_{y\in\{1,i\}} \begin{bmatrix} 0 & y \\ y^* & 0 \end{bmatrix} \otimes \underbrace{\left(\pi \cdot \text{Stab}(y^* B_{\text{high}} \cdot \left|v_1\right\rangle) \cap \pi^{-1} \cdot \text{Stab}(\left|v_0\right\rangle)\right)}_{\text{isomorphism set}} \end{aligned} \tag{3.20}$$

where π denotes a single isomorphism $y^* B_{\text{high}} \left|v_1\right\rangle \to \left|v_0\right\rangle$.

Given generating sets for $\mathrm{Stab}(v_0)$ and $\mathrm{Stab}(v_1)$, evaluating eq. (3.20) requires us to:

- **Compute $\mathrm{Stab}(A\,|w\rangle)$ from $\mathrm{Stab}(w)$ (as generating sets) for Pauli LIM A and node w.** It is straightforward to check that $\{AgA^\dagger \mid g \in G\}$, with $\langle G\rangle = \mathrm{Stab}(w)$, is a generating set for $\mathrm{Stab}(A\,|w\rangle)$.

- **Find a single isomorphism between two edges, pointing to reduced nodes.** In a reduced LIMDD, edges represent isomorphic states if and only if they point to the same nodes. This results in a straightforward algorithm, see Algorithm 16.

- **Find the intersection of two stabilizer subgroups, represented as generating sets G_0 and G_1 (IntersectStabilizerGroups, Algorithm 15).** First, it is straightforward to show that the intersection of two stabilizer subgroups is again a stabilizer subgroup (namely, it is abelian and does not contain $-\mathbb{I}$. It is never empty since $\mathbb{I}$ is a stabilizer of all states).[‖] The algorithm proceeds in two steps: first, we compute the intersection of G_0 and G_1 modulo phase; second, we "correct for" the fact that the phases play a role.

  Very broadly speaking, we use the following algebraic properties of Pauli groups. First, when considering a Pauli string λP modulo phase, it is convenient to think of it as simply the Pauli string P with phase equal to $+1$. This allows us to take an element $P \in \overline{G_0}$ from a Pauli stabilizer group $\overline{G_0}$ modulo phase, and, by abuse of language, multiply it by a phase $\lambda \in \mathbb{C}$ to obtain $\lambda \cdot P \in G_0$. Second, for any Pauli string $P \in \overline{G_0} \cap \overline{G_1}$, i.e., in the group modulo phase, there exists a unique λ such that $\lambda \cdot P \in \langle G_0\rangle$; and a unique ω such that $\omega \cdot P \in \langle G_1\rangle$. Moreover, we have $\omega = \pm\lambda$ in this case due to anti-commutativity. Consequently, if $S = \{P_1, \ldots, P_\ell\}$ is a generating set for the group $\langle S\rangle = \langle \overline{G_0}\rangle \cap \langle \overline{G_1}\rangle$ modulo phase, then we can divide these generators P_j into two sets, $S = S_0 \cup S_1$, where each $P \in S_0$ satisfies $\lambda P \in G_0 \cap G_1$ for some $\lambda \in \mathbb{C}$ and each $P \in S_1$ satisfies $\lambda P \in G_0$ and $-\lambda P \in G_1$. The algorithm finds these sets S_0 and S_1, including phase, in the loop in Line 5-11. Given such sets S_0, S_1, any element in $G_0 \cap G_1$ (i.e., the set we are interested in) can be written as a product of elements of S_0 and an *even number* of elements from S_1. The set $\{e_1 \cdot e_2, \ldots, e_1 \cdot e_m\}$, found by

[‖] To be clear, here we consider the stabilizers including their phase, i.e., we are not considering the groups modulo phase. Indeed, computing the intersection of two groups modulo phase is relatively easy, as shown in Sec. 3.5.2.

the algorithm in Line 14-16, generates precisely this set of elements generated from an even number of elements of S_1.

All of the above steps can be performed in $\mathcal{O}(n^3)$ time, where n is the number of qubits. In particular, a generating set for the intersection of G_0 and G_1 modulo phase is simply the intersection of two vector spaces over $\mathbb{F}_2$, which is constructed in time $\mathcal{O}(n^3)$ on Line 3 using the Zassenhaus algorithm. On line Line 7, checking whether $\lambda P \in G_0$ for given λ, P can be done in $\mathcal{O}(n^2)$ time; this happens at most $\mathcal{O}(|S|) = \mathcal{O}(n)$ times, so in total this operation takes up $\mathcal{O}(n^3)$ time. Lastly, the loop in Line 14-16 runs in $\mathcal{O}(n^2)$ time, as there are at most $|S| - 1 = \mathcal{O}(n)$ multiplications of Pauli strings, each of which takes $\mathcal{O}(n)$ time. We remark that the Zassenhaus algorithm cannot be straightforwardly applied to find the intersection of the groups G_0 and G_1 directly, since the elements of G_0 may not commute with those of G_1.

- `IntersectIsomorphismSets`: **Find the intersection of two isomorphism sets, represented as single isomorphism (π_0, π_1) with a generator set of a stabilizer subgroup (G_0, G_1), see Lemma 3.1.** This is the *coset intersection problem* for the PauliLIM$_n$ group. Isomorphism sets are coset of stabilizer groups (see Lemma 3.1) and it is not hard to see that that the intersection of two cosets, given as isomorphisms $\pi_{0/1}$ and generator sets $G_{0/1}$, is either empty, or a coset of $\langle G_0 \rangle \cap \langle G_1 \rangle$ (this intersection is computed using Algorithm 15). Therefore, we only need to determine an isomorphism $\pi \in \pi_0 \langle G_0 \rangle \cap \pi_1 \langle G_1 \rangle$, or infer that no such isomorphism exists.

  We solve this problem in $O(n^3)$ time in two steps (see Algorithm 14 for the full algorithm). First, we note that that $\pi_0 \langle G_0 \rangle \cap \pi_1 \langle G_1 \rangle = \pi_0 [\langle G_0 \rangle \cap (\pi_0^{-1} \pi_1) \langle G_1 \rangle]$, so we only need to find an element of the coset $S := \langle G_0 \rangle \cap (\pi_0^{-1} \pi_1) \langle G_1 \rangle$. Now note that S is nonempty if and only if there exists $g_0 \in \langle G_0 \rangle, g_1 \in \langle G_1 \rangle$ such that $g_0 = \pi_0^{-1} \pi_1 g_1$, or, equivalently, $\pi_0^{-1} \pi_1 \cdot g_1 \cdot g_0^{-1} = \mathbb{I}$. We show in Lemma 3.2 that such g_0, g_1 exist if and only if $\mathbb{I}$ is the smallest element in the set $S \pi_0^{-1} \pi_1 \langle G_1 \rangle \cdot \langle G_0 \rangle$. Hence, for finding out if S is empty we may invoke the LexMin algorithm we have already used before in GetLabels and we will explain below in Sec. 3.5.4. If it is not empty, then we obtain g_0, g_1 as above using ArgLexMin, and output $\pi_0 \cdot g_0$ as an element in the intersection. Since Lexmin and ArgLexMin take $O(n^3)$ time, so does Algorithm 14.

Lemma 3.2. The coset $S := \langle G_0 \rangle \cap \pi_1^{-1} \pi_0 \cdot \langle G_1 \rangle$ is nonempty if and only

if the lexicographically smallest element of the set $S = \pi_0^{-1}\pi_1 \langle G_1 \rangle \cdot \langle G_0 \rangle = \left\{\pi_0^{-1}\pi_1 g_1 g_0 \mid g_0 \in G_0, g_1 \in G_1\right\}$ is $1 \cdot \mathbb{I}$.

Proof. (Direction $\Rightarrow$) Suppose that the set $\langle G_0 \rangle \cap \pi_0^{-1}\pi_1 \langle G_1 \rangle$ has an element a. Then $a = g_0 = \pi_0^{-1}\pi_1 g_1$ for some $g_0 \in \langle G_0 \rangle, g_1 \in \langle G_1 \rangle$. We see that $\mathbb{I} = \pi_0^{-1}\pi_1 g_1 g_0^{-1} \in \pi_0^{-1}\pi_1 \langle G_1 \rangle \cdot \langle G_0 \rangle$, i.e., $\mathbb{I} \in S$. Note that $\mathbb{I}$ is, in particular, the lexicographically smallest element, since its check vector is the all-zero vector $(\vec{0}|\vec{0}|00)$.

(Direction $\Leftarrow$) Suppose that $\mathbb{I} \in \pi_0^{-1}\pi_1 \langle G_1 \rangle \cdot \langle G_0 \rangle$. Then $\mathbb{I} = \pi_0^{-1}\pi_1 g_1 g_0$, for some $g_0 \in \langle G_0 \rangle, g_1 \in \langle G_1 \rangle$, so we get $g_0^{-1} = \pi_0^{-1}\pi_1 g_1 \in \langle G_0 \rangle \cap \pi_0^{-1}\pi_1 \langle G_1 \rangle$, as promised. □

The four algorithms above allow us to evaluate each of the four individual terms in eq. (3.20). To finish the evaluation of eq. (3.20), one would expect that it is also necessary that we find the union of isomorphism sets. However, we note that if πG is an isomorphism set, with π an isomorphism and G an stabilizer subgroup, then $P_n \otimes (\pi g) = (P_n \otimes \pi)(\mathbb{I} \otimes g)$ for all $g \in G$. Therefore, we will evaluate eq. (3.20), i.e. find (a generating set) for all stabilizers of node v in two steps. First, we construct the generating set for the first term, i.e. $\mathbb{I} \otimes (\mathrm{Stab}(|v_0\rangle) \cap \mathrm{Stab}(B_{\mathrm{high}}|v_1\rangle))$, using the algorithms above. Next, for each of the other three terms $P_n \otimes (\pi G)$, we add only *a single* stabilizer of the form $P_n \otimes \pi$ for each $P_n \in \{X, Y, Z\}$. We give the full algorithm in Algorithm 13 and prove its efficiency below.

Lemma 3.3 (Efficiency of function `GetStabilizerGenSet`)**.** Let v be an n-qubit node. Assume that generator sets for the stabilizer subgroups of the children v_0, v_1 are known, e.g., by an earlier call to `GetStabilizerGenSet`, followed by caching the result (see Line 28 in Algorithm 13). Then Algorithm 13 (function `GetStabilizerGenSet`), applied to v, runs in time $O(n^3)$.

Proof. If $n = 1$ then Algorithm 13 only evaluates Line 2–4, which run in constant time. For $n > 1$, the algorithm performs a constant number of calls to `GetIsomorphism` (which only multiplies two Pauli LIMs and therefore runs in time $O(n)$) and four calls to `IntersectIsomorphismSets`. Note that the function `IntersectIsomorphismSets` from Algorithm 14 invokes $O(n^3)$-runtime external algorithms:

- the Zassenhaus algorithm [212] to calculate a basis for the intersection of two subspaces of a vector space,

- the RREF algorithms mentioned in Sec. 3.5.2, and
- Algorithm 2 from [125] to synthesize a circuit that transforms any stabilizer state to a basis state. Specifically, this algorithm receives as input a stabilizer subgroup G and outputs a Clifford circuit U such that $UGU^\dagger = \{Z_1, \ldots, Z_{|G|}\}$. We remark that García et al. assume in their work that G is the stabilizer group of a stabilizer state, i.e., $|G| = n$, but in fact the algorithm works also without that assumption, i.e., in the more general case when G is any abelian group of Pauli operators not containing $-\mathbb{I}$. Our algorithms use this more general use case.

Therefore, `GetStabilizerGenSet` has runtime is $O(n^3)$. □

3.5.4 Efficiently finding a minimal LIM by multiplying with stabilizers

Here, we give $O(n^3)$ subroutines solving the following problem: given generators sets G_0, G_1 of stabilizer subgroups on n qubits, and an n-qubit Pauli LIM A, determine $\min_{(g_0,g_1)\in\langle G_0,G_1\rangle} A \cdot g_0 \cdot g_1$, and also find the g_0, g_1 which minimize the expression. We give an algorithm for finding both the minimum (LexMin) and the arguments of the minimum (ArgLexMin) in Algorithm 17. The intuition behind the algorithms are the following two steps: first, the lexicographically minimum Pauli LIM *modulo scalar* can easily be determined using the scalar-ignoring DivisionRemainder algorithm from Sec. 3.5.2. Since in the lexicographic ordering, the scalar is least significant (Sec. 3.5.2), the resulting Pauli LIM has the same Pauli string as the the minimal Pauli LIM *including scalar*. We show below in Lemma 3.4 that if the scalar-ignoring minimization results in a Pauli LIM λP, then the only other eligible LIM, if it exists, is $-\lambda P$. Hence, in the next step, we only need to determine whether such LIM $-\lambda P$ exists and whether $-\lambda < \lambda$; if so, then $-\lambda P$ is the real minimal Pauli LIM $\in \langle G_0 \cup G_1\rangle$.

Lemma 3.4. Let v_0 and v_1 be LIMDD nodes, R a Pauli string and $\nu, \nu' \in \mathbb{C}$. Define $G = \mathrm{Stab}(v_0) \cup \mathrm{Stab}(v_1)$. If $\nu R, \nu' R \in \langle G\rangle$, then $\nu = \pm\nu'$.

Proof. We prove $g \in \langle G\rangle$ and $\lambda g \in \langle G\rangle$ implies $\lambda = \pm 1$. Since Pauli LIMs commute or anticommute, we can decompose both g and λg as $g = (-1)^x g_0 g_1$ and $\lambda g = (-1)^y h_0 h_1$ for some $x, y \in \{0, 1\}$ and $g_0, h_0 \in \mathrm{Stab}(v_0)$ and $g_1, h_1 \in \mathrm{Stab}(v_1)$. Combining these

Algorithm 13 Algorithm for constructing the Pauli stabilizer subgroup of a Pauli-LIMDD node. The algorithm always returns a set in reduced row echelon form (see Sec. 3.5.2), which is accomplished in line 27. In particular, the set always returns at most n elements for n-qubit states.

1: **procedure** `GetStabilizerGenSet`(EDGE $e_0 \xrightarrow{\mathbb{I}^{\otimes n}} (v_0), e_1 \xrightarrow{B_{\text{high}}} (v_1)$ **with** v_0, v_1 reduced)
2: **if** n=1 **then**
3: **if** there exists $P \in \pm 1 \cdot \{X, Y, Z\}$ **such that** $P|v\rangle = |v\rangle$ **then return** P
4: **else return** `None`
5: **else**
6: **if** $v \in$ STABCACHE$[v]$ **then return** STABCACHE$[v]$
7: $G_0 := \texttt{GetStabilizerGenSet}(v_0)$
8: **if** $B_{\text{high}} = 0$ **then**
9: **return** $\{\mathbb{I}_2 \otimes g \mid g \in G_0\} \cup \{Z \otimes \mathbb{I}^{\otimes n-1}\}$
10: **else**
11: $G := \emptyset$
12: $G_1 := \left\{ B_{\text{high}} \cdot g \cdot B_{\text{high}}^\dagger \mid g \in \texttt{GetStabilizerGenSet}(v_1) \right\}$
13: $(\pi, B) := \texttt{IntersectIsomorphismSets}((\mathbb{I}^{\otimes n-1}, G_0), (\mathbb{I}^{\otimes n-1}, G_1))$
14: $G := G \cup \{\mathbb{I}_2 \otimes g \mid g \in B\}$ ▷ Add all stabilizers of the form $\mathbb{I} \otimes \ldots$
15:
16: $\pi_0, \pi_1 := \mathbb{I}^{\otimes n-1}, \texttt{GetIsomorphism}(e_1, -1 \cdot e_1)$
17: $(\pi, B) := \texttt{IntersectIsomorphismSets}((\pi_0, G_0), (\pi_1, G_1))$
18: **if** $\pi \neq$ None **then** $G := G \cup \{Z \otimes \pi\}$ ▷ Add stabilizer of form $Z \otimes \ldots$
19:
20: $\pi_0, \pi_1 := \texttt{GetIsomorphism}(e_0, e_1), \texttt{GetIsomorphism}(e_1, e_0))$
21: $(\pi, B) := \texttt{IntersectIsomorphismSets}((\pi_0, G_0), (\pi_1, G_1))$
22: **if** $\pi \neq$ None **then** $G := G \cup \{X \otimes \pi\}$ ▷ Add stabilizer of form $X \otimes \ldots$
23:
24: $\pi_0, \pi_1 := \texttt{GetIsomorphism}(e_0, -i \cdot e_1), \texttt{GetIsomorphism}(-i \cdot e_1, e_0))$
25: $(\pi, B) := \texttt{IntersectIsomorphismSets}((\pi_0, G_0), (\pi_1, G_1))$
26: **if** $\pi \neq$ None **then** $G := G \cup \{Y \otimes \pi\}$ ▷ Add stabilizer of form $Y \otimes \ldots$
27: $G := RREF(G)$ ▷ Bring G to reduced row echelon form, potentially pruning some rows
28: STABCACHE$[v] := G$
29: **return** G

Algorithm 14 An $O(n^3)$ algorithm for computing the intersection of two sets of isomorphisms, each given as single isomorphism with a stabilizer subgroup (see Lemma 3.1).

output: Pauli LIM π, stabilizer subgroup generating set G s.t. $\pi\langle G\rangle = \pi_0\langle G_0\rangle \cap \pi_1\langle G_1\rangle$

procedure IntersectIsomorphismSets(stabilizer subgroup generating sets G_0, G_1, Pauli-LIMs π_0, π_1)
  $\pi := LexMin(G_0, G_1, \pi_1^{-1}\pi_0)$
  if $\pi = \mathbb{I}$ **then**
    $(g_0, g_1) = ArgLexMin(G_0, G_1, \pi_1^{-1}\pi_0)$
    $\pi := \pi_0 \cdot g_0$
    $G := IntersectStabilizerGroups(G_0, G_1)$
    return (π, G)
  else
    return `None`

Algorithm 15 Algorithm for finding the intersection $\langle G_0\rangle \cap \langle G_1\rangle$ of the groups generated by two stabilizer subgroup generating sets G_0 and G_1.

output: a generating set for $\langle G_0\rangle \cap \langle G_1\rangle$

procedure IntersectStabilizerGroups(stabilizer subgroup generating sets G_0, G_1)
  Use the Zassenhaus algorithm to compute the intersection modulo phase
  $S :=$ IntersectGroupsModuloPhase(G_0, G_1)
  $J, S_0, S_1 := \emptyset$
  for $P \in S$ **do**
    By abuse of language, we treat P as a Pauli string with phase $+1$
    Find $\lambda \in \{\pm 1, \pm i\}$ such that $\lambda \cdot P \in \langle G_0\rangle$
    if $\lambda \cdot P \in \langle G_1\rangle$ **then**
      $S_0 := S_0 \cup \{\lambda \cdot P\}$
    else if $-\lambda \cdot P \in \langle G_1\rangle$ **then**
      $S_1 := S_1 \cup \{\lambda \cdot P\}$
  $J := S_0$
  if $\exists e \in S_1$ **then**
    for $e' \in S_1 \setminus \{e\}$ **do**
      $q := e' \cdot e$
      $J := J \cup \{q\}$
  return J

Algorithm 16 Algorithm for constructing a single isomorphism between the quantum states represented by two Pauli-LIMDD edges, each pointing to a canonical node.

1: **procedure** `GetIsomorphism`(EDGE $\xrightarrow{A}\,(v)$, $\xrightarrow{B}\,(w)$ **with** v, w reduced, $A \neq 0 \vee B \neq 0$)
2: | **if** $v = w$ **and** $A, B \neq 0$ **then**
3: | | **return** $B \cdot A^{-1}$
4: | **return** `None`

yields $\lambda(-1)^x g_0 g_1 = (-1)^y h_0 h_1$. We recall that, if $g \in \mathrm{Stab}(|\varphi\rangle)$ is a stabilizer of any state, then $g^2 = \mathbb{I}$. Therefore, squaring both sides of the equation, we get $\lambda^2 (g_0 g_1)^2 = (h_0 h_1)^2$, so $\lambda^2 \mathbb{I} = \mathbb{I}$, so $\lambda = \pm 1$. □

The central procedure in Algorithm 17 is ArgLexMin, which, given a LIM A and sets G_0, G_1 which generate stabilizer groups, finds $g_0 \in \langle G_0 \rangle, g_1 \in \langle G_1 \rangle$ such that $A \cdot g_0 \cdot g_1$ reaches its lexicographic minimum over all choices of g_0, g_1. It first performs the scalar-ignoring minimization (Line 5) to find g_0, g_1 modulo scalar. The algorithm LexMin simply invokes ArgLexMin to get the arguments g_0, g_1 which yield the minimum and uses these to compute the actual minimum.

The subroutine FindOpposite finds an element $g \in G_0$ such that $-g \in G_1$, or infers that no such g exists. It does so in a similar fashion as IntersectStabilizerGroups from Sec. 3.5.3: by conjugation with a suitably chosen unitary U, it maps G_1 to $\{Z_1, Z_2, ..., Z_{|G_1|}\}$. Analogously to our explanation of IntersectStabilizerGroups, the group generated by $UG_1U^\dagger$ contains precisely all Pauli LIMs which satisfy the following three properties: (i) the scalar is 1; (ii) its Pauli string has an $\mathbb{I}$ or Z at positions $1, 2, ..., |G_1|$; (iii) its Pauli string has an $\mathbb{I}$ at positions $|G_1|+1, ..., n$. Therefore, the target g only exists if there is a LIM in $\langle UG_0U^\dagger \rangle$ which (i') has scalar -1 and satisfies properties (ii) and (iii). To find such a g, we put $UG_0U^\dagger$ in RREF form and check all resulting generators for properties (i'), (ii) and (iii). (By definition of RREF, it suffices to check only the generators for this property) If a generator h satisfies these properties, we return $U^\dagger h U$ and `None` otherwise. The algorithm requires $O(n^3)$ time to find U, the conversion $G \mapsto UGU^\dagger$ can be done in time $O(n^3)$, and $O(n)$ time is required for checking each of the $O(n^2)$ generators. Hence the runtime of the overall algorithm is $O(n^3)$.

Algorithm 17 Algorithms LexMin and ArgLexMin for computing the minimal element from the set $A \cdot \langle G_0 \rangle \cdot \langle G_1 \rangle = \{Ag_0g_1 \mid g_0 \in G_0, g_1 \in G_1\}$, where A is a Pauli LIM and G_0, G_1 are generating sets for stabilizer subgroups. The algorithms make use of a subroutine FindOpposite for finding an element $g \in \langle G_0 \rangle$ such that $-g \in \langle G_1 \rangle$. A canonical choice for the Rootlabel (see Sec. 3.3.3.3) of an edge e pointing to a node v is LexMin$(G, \{\mathbb{I}\}, \mathsf{label}(e))$ where G is a stabilizer generator group of Stab(v).

procedure LexMin(stabilizer subgroup generating sets G_0, G_1 and Pauli LIM A)
  output: $\min_{(g_0,g_1)\in\langle G_0 \cup G_1\rangle} A \cdot g_0 \cdot g_1$
  $(g_0, g_1) :=$ ArgLexMin(G_0, G_1, A)
  return $A \cdot g_0 \cdot g_1$

procedure ArgLexMin(stabilizer subgroup generating sets G_0, G_1 and Pauli LIM A)
output: $\arg\min_{g_0\in G_0, g_1 \in G_1} A \cdot g_0 \cdot g_1$
  $(g_0, g_1) := \underset{(g_0,g_1)\in\langle G_0 \cup G_1\rangle}{\arg\min} \{h \mid h \propto A \cdot g_0 \cdot g_1\}$
    ▷ Using the scalar-ignoring DivisionRemainder algorithm from Sec. 3.5.2,
  $g' :=$ FindOpposite(G_0, G_1, g_0, g_1)
  if g' is `None` **then**
    return (g_0, g_1)
  else
    $h_0, h_1 := g_0 \cdot g', (-g') \cdot g_1$ ▷ $g_0g_1 = -h_0h_1$
    if $A \cdot h_0 \cdot h_1 <_{\text{lex}} A \cdot g_0 \cdot g_1$ **then return** (h_0, h_1)
    else return (g_0, g_1)

procedure FindOpposite(stabilizer subgroup generating sets G_0, G_1)
output: $g \in G_0$ such that $-g \in G_1$, or `None` if no such g exists
  Compute U s.t. $UG_1U^\dagger = \{Z_1, Z_2, ..., Z_{|G_1|}\}$, using Algorithm 2 from [125]
    ▷ Z_j is the Z gate applied to qubit with index j
  $H_0 := UG_0U^\dagger$
  $H_0^{RREF} :=$ H_0 in RREF form
  for $h \in H_0^{RREF}$ **do**
    if h satisfies all three of the following: (i) h has scalar -1; the Pauli string of h (ii) contains only $\mathbb{I}$ or Z at positions $1, 2, ..., |G_1|$, and (iii) only $\mathbb{I}$ at positions $|G_1| + 1, ..., n$ **then**
      return $U^\dagger hU$
  return `None`

3.6 Numerical search for the stabilizer rank of Dicke states

Given the separation between the Clifford + T simulator —a specific stabilizer-rank based simulator— and Pauli-LIMDDs, it would be highly interesting to theoretically compare Pauli-LIMDDs and general stabilizer-rank simulation. However, proving an exponential separation would require us to find a family of states for which we can prove its stabilizer rank scales exponentially, which is a major open problem. Instead, we here take the first steps towards a numerical comparison by choosing a family of circuits which Pauli-LIMDDs can efficiently simulate and using Bravyi et al.'s heuristic algorithm for searching the stabilizer rank of the circuits' output states [64]. If the stabilizer rank is very high (specifically, if it grows superpolynomially in the number of qubits), then we have achieved the goal of showing a separation. We cannot use W states for showing this separation because the n-qubit W state $|W_n\rangle$ has linear stabilizer rank, since it is a superposition of only n computational basis states. Instead we focus on their generalization, Dicke states $|D^n_w\rangle$, which are equal superpositions of all n-qubit computational-basis status with Hamming weight w (note $|W_n\rangle = |D^n_1\rangle$),

$$|D^n_w\rangle = \frac{1}{\sqrt{\binom{n}{w}}} \sum_{x:|x|=w} |x\rangle \tag{3.21}$$

We implemented the algorithm by Bravyi et al.: see [305] for our open-source implementation. Unfortunately, the algorithm's runtime grows significantly in practice, which we believe is due to the fact that it acts on sets of quantum state vectors, which are exponentially large in the number of qubits. Our implementation allowed us to go to at most 9 qubits using the SURF supercomputing cluster. We believe this is a limitation of the algorithm and not of our implementation, since Bravyi et al. do not report beyond 6 qubits while Calpin uses the same algorithm and reaches at most 10 qubits [76]. Table 3.2 shows the heuristically found stabilizer ranks of Dicke states with our implementation. Although we observe the maximum found rank over w to grow quickly in n, the feasible regime (i.e. up to 9 qubits) is too small to draw a firm conclusion on the stabilizer ranks' scaling.

Since our heuristic algorithm finds only an upper bound on the stabilizer rank, and not a lower bound, by construction we cannot guarantee any statement on the scaling of the rank itself. However, our approach could have found only stabilizer decomposi-

tions of very low rank, thereby providing evidence that Dicke states have very slowly growing rank, meaning that stabilizer-rank methods can efficiently simulate circuits which output Dicke states. This is not what we observe; at the very least we can say that, if Dicke states have low stabilizer rank, then the current state-of-the-art method by Bravyi et al. does not succeed in finding the corresponding decomposition. Further research is needed for a conclusive answer.

We now explain the heuristic algorithm by Bravyi et al. [64], which has been explained in more detail in [76]. The algorithm follows a simulated annealing approach: on input n, w and χ, it performs a random walk through sets of χ stabilizer states. It starts with a random set V of χ stabilizer states on n qubits. In a single 'step', the algorithm picks one of these states $|\psi\rangle \in V$ at random, together with a random n-qubit Pauli operator P, and replaces the state $|\psi\rangle$ with $|\psi'\rangle := c(\mathbb{I} + P)|\psi\rangle$ with c a normalization constant (or repeats if $|\psi'\rangle = 0$), yielding a new set V'. The step is accepted with certainty if $F_V < F_{V'}$, where $F_V := |\langle D_w^n| \Pi_V |D_w^n\rangle|$ with Π_V the projector on the subspace of the n-qubit Hilbert space spanned by the stabilizer states in V. Otherwise, it is accepted with probability $\exp(-\beta(F_{V'} - F_V))$, where β should be interpreted as the inverse temperature. The algorithm terminates if it finds $F_V = 1$, implying that $|D_w^n\rangle$ can be written as linear combination of V, outputting the number χ as (an upper bound on) the stabilizer rank of $|\psi\rangle$. For a fixed χ, we use identical values to Bravyi et al. [64] and vary β from 1 to 4000 in 100 steps, performing 1000 steps at each value of β.

3.7 Related work

We mention related work on classical simulation formalisms and decision diagrams other than QMDDs.

The Affine Algebraic Decision Diagram, introduced by Tafertshofer and Pedam [308], and by Sanner and McAllister [281], is akin to a QMDD except that its edges are labeled with a pair of real numbers (a, b), so that an edge $\xrightarrow{(a,b)}\!\!\textcircled{v}$ represents the state vector $a|v\rangle + b|+\rangle^{\otimes n}$ (i.e., here b is added to each element of the vector $a|v\rangle$). To the best of our knowledge, this diagram has not been applied to quantum computing.

Context-Free-Language Ordered Binary Decision Diagrams (CFLOBDDs) [293, 294] extend BDDs with insights from visibly pushdown automata [10]. An extension of

Table 3.2: Heuristically-found upper bounds on the stabilizer rank χ of Dicke states $|D_w^n\rangle$ (eq. (3.21)) using the heuristic algorithm from Bravyi et al. [64], see text in Section 3.6 for details. We investigated up to 9 qubits using the SURF supercomputing cluster (approximately the same as the number of qubits reached in the literature as described in the text). Empty cells indicate non-existing or not-investigated states. In particular, we have not investigated $w > \lfloor \frac{n}{2} \rfloor$ since $|D_w^n\rangle$ and $|D_{n-w}^n\rangle$ have identical stabilizer rank because $X^{\otimes n} |D_w^n\rangle = |D_{n-w}^n\rangle$. For $|D_3^8\rangle$ and $|D_4^9\rangle$, we have run the heuristic algorithm to find sets of stabilizers up to size 11 (theoretical upper bound) and 10, respectively, but the algorithm has not found sets in which these two Dicke states could be decomposed. We emphasize that the algorithm is heuristic, so even if there exists a stabilizer decomposition of a given rank, the algorithm might not find it.

	Hamming weight w				
#qubits n	**0**	**1**	**2**	**3**	**4**
1	1				
2	1	1			
3	1	2			
4	1	2	2		
5	1	3	2		
6	1	3	4	2	
7	1	4	7	4	
8	1	4	8	≤ 11	5
9					> 10?

CFLOBDD to the complex domain [296] shows good performance for various simulation of quantum computing benchmarks. Sentential Decision Diagrams [96] generalize BDDs by replacing their total variable order with a variable tree (vtree). Although Kisa et al. [180] introduced an SDD which represents probability distributions, SDDs have not yet been used to simulate quantum computing, to the best of our knowledge. The Variable-Shift SDD (VS-SDD) [235] improves on the SDD by merging isomorphic vtree nodes. We remark that CFLOBDDs are similar to VS-SDD with a balanced vtree.

Günther and Drechsler introduced a BDD variant [146] which, in LIMDD terminology, has a label on the root node only. To be precise, this diagram's root edge is labeled with an invertible matrix $A \in \mathbb{F}_2^{n \times n}$. If the root node represents the function $r \colon \mathbb{B}^n \to \mathbb{B}$, then the diagram represents the function $f(\vec{x}) = r(A \cdot \vec{x})$. (This concepts extends trivially to the domain of pseudo-Boolean functions, by replacing the BDD with an ADD.) In contrast, LIMDDs allow a label on every edge in the diagram, not only the root edge. We show that this is essential to capture stabilizer states.

A multilinear arithmetic formula is a formula over $+, \times$ which computes a polynomial in which no variable appears raised to a higher power than 1. Aaronson showed that some stabilizer states require superpolynomial-size multilinear arithmetic formulas [1, 65].

3.8 Discussion

We have introduced LIMDD, a novel decision diagram-based method to simulate quantum circuits, which enables polynomial-size representation of a strict superset of stabilizer states and the states represented by polynomially large QMDDs. To prove this strict inclusion, we have shown the first lower bounds on the size of QMDDs: they require exponential size for certain families of stabilizer states. Our results show that these states are thus hard for QMDDs. We also give the first analytical comparison between simulation based on decision diagrams, and matrix product states, and the Clifford + T simulator.

LIMDDs achieve a more succinct representation than QMDDs by representing states up to local invertible maps which uses single-qubit (i.e., local) operations from a group G. We have investigated the choices $G = \textsc{Pauli}$, $G = \langle Z \rangle$ and $G = \langle X \rangle$, and found that any choice suffices for an exponential advantage over QMDDs; notably, the choice $G = \textsc{Pauli}$ allows us to succinctly represent any stabilizer state. Furthermore, we showed how to simulate arbitrary quantum circuits, encoded as Pauli-LIMDDs. The resulting algorithms for simulating quantum circuits are exponentially faster than for QMDDs in the best case, and never more than a polynomial factor slower. In the case of Clifford circuits, the simulation by LIMDDs is in polynomial time (in contrast to QMDDs).

We have shown that Pauli-LIMDDs can efficiently simulate a circuit family outputting the W states, in contrast to the Clifford + T simulator which requires exponential time to do so (assuming the widely believed ETH), even when allowing for preprocessing of the circuit with a T-count optimizer.

Since we know from experience that implementing a decision diagram framework is a major endeavor, we leave an implementation of the Pauli-LIMDD, in order to observe its runtimes in practice on relevant quantum circuits, to future work. We emphasize that from the perspective of algorithm design, we have laid all the groundwork for such

an implementation, including the key ingredient for the efficiency of many operations for existing decision diagrams: the existence of a unique canonical representative of the represented function, combined with a tractable MakeEdge algorithm to find it.

Regarding extensions of the LIMDD data structure, an obvious next step is to investigate other choices of G. Of interest are both the representational capabilities of such diagrams (do they represent interesting states?), and the algorithmic capabilities (can we still find efficient algorithms which make use of these diagrams?). In this vein, an important question is what the relationship is between G-LIMDDs (for various choices of G) and existing formalisms for the classical simulation of quantum circuits, such as those based on match gates [152,177,311] and tensor networks [163,248]. It would also be interesting to compare LIMDDs to graphical calculi such as the ZX calculus [89], following similar work for QMDDs [336].

Lastly, we note that the current definition of LIMDD imposes a strict total order over the qubits along every path from root to leaf. It is known that the chosen order can greatly influence the size of the DD [278, 344], making it interesting to investigate variants of LIMDDs with a flexible ordering, for example taking inspiration from the Sentential Decision Diagram [96, 180].