LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States

Lieuwe Vinkhuijzen∗,1, Tim Coopmans∗,1,2, David Elkouss2,3, Vedran Dunjko1, and Alfons Laarman1

1Leiden University, The Netherlands
2Delft University of Technology, The Netherlands
3Networked Quantum Devices Unit, Okinawa Institute of Science and Technology Graduate University, Okinawa, Japan

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. By uniting two successful approaches, LIMDDs thus pave the way for fundamentally more powerful solutions for simulation and analysis of quantum computing.

∗These authors contributed equally.
1 Introduction

Classical simulation of quantum computing is useful for circuit design [1, 2], verification [3, 4] and studying noise resilience in the era of Noisy Intermediate-Scale Quantum (NISQ) computers [5]. 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 [6–10]. Stabilizer states, and associated formalisms for expressing them, are fundamental to many quantum error correcting codes [8] and play a role in measurement-based quantum computation [11]. In fact, simulation of universal quantum circuits is fixed-parameter tractable in the number of non-Clifford gates [12], which is why many modern simulators are based on stabilizer decomposition [12–17].

Another method for simulating universal quantum computation is based on decision diagrams (DDs) [18–21], including Algebraic DDs [22–25], Affine Algebraic DDs [26], Quantum Multi-valued DDs [27, 28], and Tensor DDs [29]. 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 homomorphic (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 [27–29]. Indeed, DD-based simulation was empirically shown to be competitive with state-of-the-art simulators [21, 28, 30] and is used in several simulator and circuit verification implementations [31, 32]. DDs and the stabilizer formalism are introduced in Sec. 2.

QMDDs are currently the most succinct DD supporting quantum simulation, but in this paper we show that they require exponential size to represent a type of stabilizer state called a cluster state [33]. 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 [34, 35], in Sec. 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) [36, 38]. Fig. 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 [39] and quantum [2] 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) [40–42]. We provide algorithms for both in Sec. 4. Indeed, the main technical contribution of this paper 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 $\varphi$, $\psi$, 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 $P = P_n \otimes \cdots \otimes P_1$ 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., $|\varphi\rangle = c_0 |0\rangle + c_1 |1\rangle |\varphi_1\rangle$ for complex coefficients $c_0, c_1$. We then realize that $P_{\text{rest}} = P_{n-1} \cdots \otimes P_1$ 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 anti-diagonal, respectively). Hence $P_{\text{rest}}$ 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 $\pi$ is a single isomorphism $|\varphi_0\rangle \rightarrow |\psi_0\rangle$.

Thus, to find a (canonical) isomorphism between $|\varphi\rangle$, $|\psi\rangle$, we need three algorithms: to find (a) an isomorphism between $n$-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 $|\varphi\rangle \rightarrow |\psi\rangle$, we select the lexicographical minimum to ensure canonicity. We emphasize that the algorithm works for arbitrary quantum states, not only stabilizer states.

2 Preliminaries

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).

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

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

The Quantum Multi-valued Decision Diagram (QMDD) \cite{27} is a data structure which can succinctly represent functions of the form $f : \{0, 1\}^n \rightarrow \mathbb{C}$, and thus can represent any quantum state per Eq. 1. A QMDD is a rooted DAG with a unique leaf node $\text{L}$ representing the value 1. Fig. 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 $f(x_n, \ldots, x_1)$ 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., $f(1, 1, 0) = \frac{1}{2} \cdot 1 \cdot -\sqrt{2} \cdot 1 = -\frac{\sqrt{2}}{4}$ in Fig. 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 $f_v (x_k, \ldots, x_1) \triangleq f(a_n, \ldots, a_{k-1}, x_k, \ldots, x_1)$. The node $v$ represents this subfunction up to a complex factor $\gamma$, which is stored on the edge incoming to $v$ along that path. This allows any two nodes which represent functions equal up
2.2 Pauli operators and stabilizer states

In contrast to decision diagrams, the stabilizer formalism [6] 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 \( \text{PAULI} \triangleq \{I, X, Y, Z\} \) as defined
states that is also important in error correction and measurement-based quantum computing [44].

Given a vector space $V^\otimes n$ by only $n$-qubit Pauli strings $P_n \otimes \cdots \otimes P_1$ with $P_i \in \text{PAULI}$. Here we used the notation $(G) = H$ to denote that $G \subseteq H$ is a generator set for a group $H$. One can check that $\text{PAULI}_n = \{\mathbb{I}P_n \otimes \cdots \otimes P_1 \mid P_1, \ldots, P_n \in \text{PAULI}, c \in \{0, 1, 2, 3\}\}$, so in particular we have $\text{PAULI}_1 = \{\pm P, \pm iP \mid P \in \text{PAULI}\}$ (the Pauli set with a factor $\pm 1$ or $\pm i$).

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

The set of Pauli stabilizers $\text{Stab}(\varphi) \subset \text{PAULI}_n$ of an $n$-qubit quantum state $\varphi$ necessarily forms a subgroup of $\text{PAULI}_n$, since the identity operator $\mathbb{I}^\otimes n$ is a stabilizer of any $n$-qubit state and moreover if $U$ and $V$ stabilize $\varphi$, 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 $G$ either commute ($AB = BA$) or anticommute ($AB = -BA$) under multiplication and anticommuting elements can never be stabilizers of the same state $\varphi$, because if $A, B \in \text{Stab}(\varphi)$ 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 sufficient: the class of abelian subgroups $G$ of $\text{PAULI}_n$, 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 $\lambda$ of any stabilizer $\lambda P \in \text{PAULI}$ can only be $\lambda = \pm 1$, derived as

$$\forall \lambda P \in \text{Stab}(\varphi) : |\varphi\rangle = (\lambda P)|\varphi\rangle = (\lambda P)^2 |\varphi\rangle = \lambda^2 |\varphi\rangle \implies \lambda = \pm 1. \quad (2)$$

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:

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

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 $O(n)$ bits to encode the Pauli string (plus factor), yielding $O(n^2)$ 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 $n$ [6, 7]. Various efficient algorithms exist for manipulating stabilizer (sub)groups $S$, including testing membership (is $A \in \text{PAULI}_n$ 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 App. A 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 [43] and quantum error correction [8]. Examples of stabilizer subgroups are $\{I\}$ for $\frac{1}{\sqrt{2}}(|00\rangle + e^{i\pi/4}|11\rangle)$, $\langle Z \rangle$ for $|1\rangle$ and $\langle X \otimes X \rangle$ for $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ and $\frac{1}{\sqrt{2}}(00\rangle + |11\rangle) + \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. 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 $\text{CZ} \triangleq (|00\rangle + |01\rangle + |10\rangle + |11\rangle) - 11\langle 11\rangle$ gates, and form a strict subset of all stabilizer states that is also important in error correction and measurement-based quantum computing [44]. 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 [45]. Each coset state is a stabilizer state.
Stabilizer decomposition-based methods [12–17] extend the stabilizer formalism to families of Clifford circuits with arbitrary input states $|\varphi_n\rangle$, enabling the simulation of universal quantum computation [46]. By decomposing the $n$-qubit state $|\varphi_n\rangle$ as linear combination of $\chi$ stabilizer states followed by simulating the circuit on each of the $\chi$ stabilizer states, the measurement outcomes can be computed in time $O(\chi^2 \cdot \text{poly}(n))$, where the least $\chi$ 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 $|\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$; thus, an $n$-qubit circuit containing $t$ $T$ gates will be converted into an $n + t$-qubit Clifford circuit with input state $|\varphi\rangle = |0\rangle^\otimes n \otimes (|\rangle^{+})^\otimes t$ [12]. We will refer to the resulting specific stabilizer-rank based simulation method as the ‘Clifford + $T$ simulator,’ whose simulation runtime scales with $\chi_t = \chi((T^{+})^\otimes t)$, the number of stabilizer states in the decomposition of $|\varphi\rangle$.

Trivially, we have $\chi_t \leq 2^t$, and although recent work [12, 13] has found decompositions smaller than $2^t$ terms based on weak simulation methods, the scaling of $\chi_t$ 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).

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 [36, 37]. For qubits, an $n$-qubit MPS $M$ is formally defined as a series of $2n$ matrices $A^x_k \in \mathbb{C}^{D_k \times D_{k+1}}$ 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 $|x_1x_2...x_n\rangle |M\rangle = A^{x_1}_{1}A^{x_2}_{2}...A^{x_n}_{n}$ 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 $|\varphi\rangle = \sum_{j=1}^{m} \alpha_j |a_j\rangle_A \otimes |b_j\rangle_B$ for complex coefficients $\alpha_j$, 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 [38] showed that MPS-based circuit simulation is possible in time $O(n \cdot \text{poly}(\chi))$ per elementary operation, where $n$ is number of qubits and $\chi$ the maximum Schmidt rank for all intermediate states computed.

3 Local Invertible Map Decision Diagrams

Sec. 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. 3.2 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 by giving algorithms to update and query the LiMDD data structure. These LiMDD manipulation algorithms take a LiMDD φ, representing some state |φ⟩, and return another LiMDD ψ that represents the state |ψ⟩ = U|φ⟩ 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 quantum states, i.e., take polynomial time in the size of the 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 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 Sec. 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. 1 provides the bottom line of this section by comparing simulator runtimes. In Sec. 3.4, we prove Prop. 1.

Proposition 1. Let $QSim_{Clifford}^C + T_C$ denote the runtime of the Clifford + T simulator on circuit C (allowing for weak simulation as in [13]). Let $QSim_{T}^C$ denote the runtime of strong simulation of circuit C using method $D = \langle Pauli\rangle$ 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 $\Omega^*$ discards polynomial factors, i.e., $\Omega^*(f(n)) \triangleq \Omega(n^{O(1)}f(n))$.

There is a family of circuits C such that:

1. LiMDD is exponentially faster than Clifford + T: $QSim_{Clifford}^C + T_C = \Omega^*(2^n \cdot QSim_{LiMDD}^C)$,\(^\dagger\)
2. LiMDD is exponentially faster than MPS: $QSim_{MP}^C = \Omega^*(2^n \cdot QSim_{LiMDD}^C)$, and
3. LiMDD is exponentially faster than QMDD: $QSim_{QM}^C = \Omega^*(2^n \cdot QSim_{LiMDD}^C)$.
4. For all C, LiMDD is at worst cubically slower than QMDD: $QSim_{LiMDD}^C = O(2^n, QSim_{QM}^C)$.
5. Item 3 and 4 hold when replacing QMDD with QMDD $\cup$ Stab.

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 Def. 1). As a result, LiMDDs can be exponentially more succinct than QMDDs, for example in the case of stabilizer states (see Sec. 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 [34, App. A] and [35, 47]).

Definition 1 (G-LIM, G-Isomorphism). An n-qubit G-Local Invertible Map (LIM) is an operator $O$ of the form $O = \lambda O_1 \otimes \cdots \otimes O_n$, where G is a group of invertible 2 $\times$ 2 matrices, $O_i \in G$ and $\lambda \in C \setminus \{0\}$. A G-isomorphism between two n-qubit quantum states $|\phi\rangle, |\psi\rangle$ is a LIM $O$ such that $O |\phi\rangle = |\psi\rangle$, denoted $|\phi\rangle \approx_G |\psi\rangle$. Note that G-isomorphism is an equivalence relation.

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

\(^\dagger\)Assuming the exponential time hypothesis (ETH). See Sec. 3.4.3 for details.
We define $\text{PAULI}_n \triangleq \{\text{Pauli}\}$-LIM, i.e., the group of Pauli operators $P \in \text{PAULI}_n$ with arbitrary complex factor $\lambda \in \mathbb{C} \setminus \{0\}$ ($\lambda$ can absorb the factor $\gamma = \pm 1, \pm i$ in $P = \gamma P_n \otimes \cdots \otimes P_1$. Note $\lambda = \pm 1$ still for $\text{PAULI}_n$ operators which are stabilizers, by eq. (2)).

Before we give the formal definition of LIMDDs in Def. 2, we give a motivating example in Fig. 3, which uses $(X, Y, Z, T)$-LIMs to demonstrate how the use of 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 $|q_1\rangle = [1, 1]^{T}$ and $|q'_1\rangle = [1, -1]^{T}$, 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 $\ell_1$ 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 $\ell_2$, observing that $|q'_2\rangle = I \otimes Z |q_2\rangle$. Second, we create a node $\ell'_2$ 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 $\ell'_2$, placing these isomorphisms on the respective edges. To go from (c) to (d), we merge nodes $q_3, q'_3$ into node $\ell_3$ by noticing that $|q'_3\rangle = (Z \otimes I \otimes Z) |q_3\rangle$. This isomorphism $Z \otimes I \otimes Z$ is stored on the high edge out of the root node. We have $|q_3\rangle = I \otimes I \otimes X |\ell_3\rangle$, so we propagate the isomorphism $I \otimes I \otimes X$ upward, and store it on the root edge. Therefore, the final LIMDD has the LIM $\frac{1}{4} I \otimes I \otimes I \otimes X$ on its root edge.

The resulting data structure in Fig. 3 is a LIMDD of only six nodes instead of ten, but requires additional storage for the LIMs. Sec. 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 Fig. 3) only considers particular choices for LIMs. For instance, it would be equally valid to select $\text{LIM} I \otimes Z$ instead of $-I \otimes XZ$ 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. Sec. 4 provides a solution for $\{\text{Pauli}\}$-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 $\{I \otimes X, Y \otimes I\}$ for $q_2$.

**Definition 2.** An $n$-G-LIMDD is a rooted, directed acyclic graph (DAG) representing an $n$-qubit quantum state. Formally, it is a 6-tuple $(\text{NODE} \cup \{\text{Leaf}\}, \text{idx}, \text{low}, \text{high}, \text{label}, e_{\text{root}})$, where:

- **Leaf** (a sink) is a unique leaf node with qubit index $\text{idx}(\text{Leaf}) = 0$;
- **Node** is a set of nodes with qubit indices $\text{idx}(v) \in \{1, \ldots, n\}$ for $v \in \text{NODE}$;

---

Accepted in *Quantum* 2023-07-17, click title to verify. Published under CC-BY 4.0.
- \( e_{\text{root}} \) is a root edge without source pointing to the root node \( r \in \text{NODE} \) with \( \text{idx}(r) = n \);

- low, high : \( \text{NODE} \to \text{NODE} \cup \{ \text{Leaf} \} \) indicate the low and high edge functions, respectively. We write low, (or high,) to obtain the edge \( (v,w) \) with \( w = \text{low}(v) \) (or \( w = \text{high}(v) \)). For all \( v \in \text{NODE} \) it holds that \( \text{idx}(\text{low}(v)) = \text{idx}(\text{high}(v)) = \text{idx}(v) - 1 \) (no qubits are skipped);

- label : \( \text{low} \cup \text{high} \cup \{ e_{\text{root}} \} \to k - \text{G-LIM} \cup \{ 0 \} \) is a function labeling edges \( (.,w) \) with \( k - \text{G-LIMs} \) or 0, where \( k = \text{idx}(w) \).

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

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

\[ |\text{Leaf} \rangle \equiv 1 \]
\[ |e \rangle \equiv \text{label}(e) \cdot |v \rangle \]
\[ |v \rangle \equiv |0 \rangle \otimes |\text{low}_v \rangle + |1 \rangle \otimes |\text{high}_v \rangle \]

It follows from this definition that a node \( v \) with \( \text{idx}(v) = k \) 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 \( \nabla^A \cdot \Box^B \) or a LIMDD \( \nabla \cdot \Box \) with root edge \( \nabla \cdot \Box \).

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 \( e_0 \nabla \Box \) and \( e_1 \nabla \Box \) can be computed just like for QMDDs [27]: Take all edges \( \nabla \cdot \Box \) pointing to the leaf in the LIMDD \( e_0 \) and replace them with edges \( \alpha \nabla \Box \) 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 \( C \otimes I^m \).

We can now consider various instantiations of the above general LIMDD definition for different LIM groups \( G \). A G-LIMDD with \( G = \{1\} \) yields precisely all QMDDs by definition, i.e., all edges labels effectively only contain scalars. As all groups \( G \) contain the identity operator \( I \), 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. 3.2. Finally, and most importantly, \( \langle \text{Pauli} \rangle \)-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. 3.2.

In what follows, we only consider \( \langle X \rangle \), \( \langle Z \rangle \), and \( \langle \text{Pauli} \rangle \)-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 \( e_{\text{root}} \) points to node \( r \) and is labeled with the LIM \( \text{label}(e_{\text{root}}) = A = \lambda P_n \otimes \cdots \otimes P_1 \in \text{Pauli-LIM}_{n}\). First, we substitute \( |v \rangle = |0 \rangle |\text{low}_v \rangle + |1 \rangle |\text{high}_v \rangle \), where \( \text{low}_v, \text{high}_v \) are the low and high edges going out of \( r \), thus obtaining \( \langle x|\psi \rangle = \langle x|e_{\text{root}} \rangle = \langle x| A (|0 \rangle |\text{low}_v \rangle + |1 \rangle |\text{high}_v \rangle \). Next, we notice that \( \langle x|A = \lambda (\langle x| P_n \rangle \otimes \cdots \otimes (\langle x| P_1) \rangle = \gamma \langle y \rangle \) for some \( \gamma \in \mathbb{C} \) and a computational basis state \( |y \rangle \). Therefore, letting \( y' = y_{n-1} \cdots y_1 \), it suffices to compute \( \langle y_n (\langle y' | (|0 \rangle |\text{low}_v \rangle + |1 \rangle |\text{high}_v \rangle) \), which reduces to computing either \( \langle y' |\text{low}_v \rangle \) if \( y_n = 0 \), or \( \langle y' |\text{high}_v \rangle \) 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 \( \gamma \) found along this path. Alg. 1 formalizes this. Its runtime is \( O(n^2) \).

\footnote{Decision diagram definitions [18, 19, 48] 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 [41].}
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) [36, 37]. 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 Fig. 1.

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

Fig. 4 visualizes succinctness relations between different quantum state representations, as proved in Prop. 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, we show the strongest result, namely that LIMDDs are also more succinct than the union of QMDDs and stabilizer states, written $\text{QMDD} \cup \text{Stab}$, which can be thought of as 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 2.** The inclusions and separations in Fig. 4 hold.

**Proof.** The inclusions between the sets of states shown in gray are well known [44, 45]. The inclusions between decision diagrams hold because, e.g., a QMDD is a G-LIMDD with $G = \{1\}$, i.e., each label is of the form $\lambda |_n$ with $\lambda \in \mathbb{C}$, as discussed in Sec. 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 Th. 1 and App. C (which also shows that poly-sized LIMDD includes QMDD $\cup$ Stab). Corollary 3 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. Th. 2 shows the separation with QMDDs by demonstrating that the so-called (two-dimensional) cluster state, requires $2^{O(\sqrt{n})}$ nodes as QMDD. Finally, App. C proves the same for coset states. \(\square\)

Th. 1 shows that any stabilizer state can be represented as a $\langle \text{Pauli} \rangle$-Tower LIMDD (Def. 3).
Definition 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 $I^{\otimes k}$, 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 $\lambda \in C \setminus \{0\}$ (i.e., in contrast to high edges, the root edge can have an arbitrary scalar). Fig. 6 depicts a n-qubit G-Tower LIMDD.

Theorem 1. Let $n > 0$. Each n-qubit stabilizer state is represented up to normalization by a (Pauli)-Tower LIMDDs of Def. 3, e.g., where the scalars $\lambda$ of the Pauli LIMs $AP$ on high edges are restricted as $\lambda \in \{0, \pm 1, \pm i\}$. Conversely, every such LIMDD represents a stabilizer state.

Proof sketch of Th. 1. (Full proof in App. C) 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 (Pauli)-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 (Pauli)-Tower-LIMDD $\bigotimes \mathcal{O}_{a} \bigotimes \mathcal{O}_{\beta}$. 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 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

$$|\psi\rangle = \frac{1}{\sqrt{2}} (|0\rangle |\psi_0\rangle + \lambda |1\rangle \otimes (P_n \otimes \cdots \otimes P_1 |\psi_0\rangle))$$

Hence $|\psi\rangle$ is represented by the Tower Pauli-LIMDD $\bigotimes \mathcal{O} \bigotimes \mathcal{O} \bigotimes \mathcal{O}$. 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 $\sqrt{2} |\psi_{GHZ}\rangle = \frac{1}{\sqrt{2}} (|000\rangle + |111\rangle)$, for $|e_{++}\rangle = |+ +\rangle$ where $|+\rangle = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)$, and the state $|e_\phi\rangle = \frac{1}{\sqrt{2}} (|000\rangle - |001\rangle + i|010\rangle + i|011\rangle + i|100\rangle + i|101\rangle - |110\rangle + |111\rangle)$ with stabilizer group generators $\{X \otimes X \otimes X, -Z \otimes Z \otimes X, I \otimes Y \otimes Z\}$.

We also note that Th. 1 demonstrates that for any n-qubit stabilizer state $|\varphi\rangle$, the $(n - 1)$-qubit states $(|0\rangle \otimes I_{2^n-1}) |\varphi\rangle$ and $(|1\rangle \otimes I_{2^n-1}) |\varphi\rangle$ are not only stabilizer states, but also PauliLIM-
isomorphic. While we believe this fact is known in the community,\footnote{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 [6]. Similarly, the characterization in Prop. 2 of (Z)-Tower-LIMDDs as representing precisely the graph states, is immediate by defining graph states recursively (see App. C). The fact that (X)-Tower LIMDDs represent coset states is less evident and requires a separate proof, which we also give in App. C.} 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, Th. 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 shows that a trivial combination with stabilizer formalism does not solve this issue.

**Theorem 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 $\lceil n/3 \rceil$ vertices in $S$ that are adjacent to a vertex in $T$ [49, 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 $a \in \{0, 1\}^{\frac{3}{4}n^2}$ induce $2^{\lfloor n/12 \rfloor}$ different subfunctions $f_T$, where $f : \{0, 1\}^n \rightarrow \mathbb{C}$ is the 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. B. \hfill $\Box$

**Corollary 3** (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 $\frac{1}{\sqrt{2}}$ for $z \in \{\pm 1, \pm i\}$ and some integer $k \geq 1$ [9], while $|1\rangle \otimes (|0\rangle \oplus \mathbb{R}) |\varphi\rangle = e^{i\pi/4} \cdot \left(\frac{1}{\sqrt{2}}\right)^{n^2+1}$ is not of this form. Its canonical QMDD and Pauli-LIMDD have root nodes $\circlearrowleft_{\mathbb{R}} \otimes e^{i\pi/4} \otimes \circlearrowleft_{\mathbb{R}}$ and $\circlearrowleft_{\mathbb{R}} \otimes e^{i\pi/4} \circlearrowleft_{\mathbb{R}}$, where the respective diagram for $G_n$ is exponentially large (Th. 2) and polynomially small (Th. 1). \hfill $\Box$

### 3.2.2 LIMDDs are exponentially more succinct than matrix product states

Th. 4 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 4.** 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 $2^{D(n)}$.

**Proof.** Van den Nest et al. [50] 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 Sec. 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 [51]. This proves the theorem. \hfill $\Box$
In contrast, the Pauli-LIMDD efficiently represents cluster states, and more generally all stabilizer states (Th. 1).

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 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. 3.1, 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 \( \circledast \circledast_{A,B} \), returns \( C \circledast \) with \( C|v\rangle = |w\rangle \), where \( v \) is the unique, canonical node in the diagram that is G-isomorphic to node \( w \). Sec. 4.2 provides a \( O(n^3) \) MAKEEdge algorithm for \( (\text{Pauli}) \)-LIMDDs satisfying this specification. For now, the reader may assume the provisional implementation in Alg. 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, Alg. 3 makes any LIMDD canonical using dynamic programming and the (real) \( O(n^3) \) MAKEEdge algorithm from Sec. 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 \( \rightarrow \) EDGE stores for each node its canonical counterpart as soon as it is computed at Line 4. The algorithm therefore runs in time \( O(n^3m) \) where \( m \) is the number of nodes in the original diagram.

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 shows how we do this.

Table 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).

<table>
<thead>
<tr>
<th>Operation \ input:</th>
<th>QMDD</th>
<th>LIMDD</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single (</td>
<td>0\rangle ) / (</td>
<td>1\rangle )-basis measurement</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>Single Pauli gate</td>
<td>( O(m) )</td>
<td>( O(1) )</td>
<td>Sec. 3.3.2</td>
</tr>
<tr>
<td>Single Hadamard gate / ADD()</td>
<td>( O(2^n) ) *</td>
<td>( O(n^32^n) ) *</td>
<td>Sec. 3.3.2</td>
</tr>
<tr>
<td>Clifford gate on stabilizer state</td>
<td>( O(2^n) )</td>
<td>( O(n^4) )</td>
<td>Sec. 3.3.4</td>
</tr>
<tr>
<td>Multi-qubit gate</td>
<td>( O(4^n) )</td>
<td>( O(n^34^n) )</td>
<td>Sec. 3.3.3</td>
</tr>
<tr>
<td>MAKEEdge</td>
<td>( O(1) )</td>
<td>( O(n^3) )</td>
<td>Sec. 4.2</td>
</tr>
<tr>
<td>Checking state equality</td>
<td>( O(1) )</td>
<td>( O(n^3) )</td>
<td>Sec. 4.2.2</td>
</tr>
</tbody>
</table>

\*The worst-case of QMDDs and LIMDDs is caused by the vector addition introduced by the Hadamard gate [52, Table 2, +BC, +SLDD]. See Fig. 9 for an example.
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. 4.2) returns a canonical node, assuming $v$, $w$ are already canonical.

1: procedure MAKEEDGE(Edge $A\xrightarrow{\lambda} \bigcirc$, Edge $B\xrightarrow{\ell} \bigcirc$)
2: $u := \bigodot A \xrightarrow{\ell} \bigodot B$
3: return Edge $v \xrightarrow{\lambda} w$  $\triangleright$ Where $k = \text{id}(u)$

Algorithm 3 Make any LIIMDD canonical using MAKEEDGE.

1: procedure MAKECANONICAL(Edge $A\xrightarrow{\lambda} \bigcirc$)  $\triangleright$ Compute result once for $v$ and store in cache:
2: if $v \notin \text{Canonic} \text{aCache}$ then
3: $e_0, e_1 := \text{MAKECANONICAL}(|\text{low}(v))\rangle$, \text{MAKECANONICAL}(|\text{high}(v))\rangle
4: \text{Canonic}aCache$\text{[}v\text{]} := MAKEEDGE($e_0, e_1$)
5: return $A \cdot \text{Canonic}aCache$\text{[}v\text{]}  $\triangleright$ Retrieve result from cache

Finally, in this section, we often decompose LIIM 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 Alg. 4 to easily navigate diagrams according to edge semantics. Provided with a bit string $x_n \ldots x_1$, the procedure is the same as READAMPLITUDE. If however fewer bits are supplied, it returns a LIIM root edge representing a subvector. For instance, the subvector for $|10\rangle$ of the LIIMDD root edge $e_r$ in Fig. 3 (d) is computed by taking $|\text{FOLLOW}_{10}(e_r)\rangle = |\frac{1}{2^4} \otimes XZ\rangle = \frac{1}{4} \cdot [1, 1, -i, i]$. So, we can specify it as $|\text{FOLLOW}_{6}(e)\rangle = ((b) \otimes \mathbb{I}^{n-f}) |e\rangle$, i.e., select the $b$th block of size $2^{n-f}$ from the vector $|e\rangle$ (or rather, return a LIIMDD edge representing that block).

Algorithm 4 FOLLOW: a generalization of READAMPLITUDE, returning edges.

1: procedure FOLLOW(Edge $e \xrightarrow{\lambda P_n \otimes \cdots \otimes P_k} \bigcirc$, $x_n, \ldots, x_k \in \{0, 1\}$ with $n = \text{id}(v)$ and $k \geq 1$)
2: if $k > n$ then return $\xrightarrow{\lambda P_n \otimes \cdots \otimes P_k} \bigcirc$  $\triangleright$ End of bit string
3: $\langle y_n \ldots y_k \rangle := \langle x_n \ldots x_k \rangle A P_n \otimes \cdots \otimes P_k$  $\triangleright$ $O(n)$-computable LIIM operation
4: if $y_n = 0$ then
5: return $\gamma \cdot \text{FOLLOW}(|\text{low}(e), y_{n-1}, \ldots, y_k\rangle$  $\triangleright$ $y_n = 0$
6: else
7: return $\gamma \cdot \text{FOLLOW}(|\text{high}(e), y_{n-1}, \ldots, y_k\rangle$  $\triangleright$ $y_n = 1$

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. E gives general algorithms with the same worst-case runtimes.

The procedure MEASUREMENTPROBABILITY in Alg. 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 $e_0 = \text{FOLLOW}_n(e)$ using the SQUAREDNORM subroutine. The total runtime is dominated by the subroutine SQUAREDNORM, which computes the quantity $\langle e|e\rangle$ given a LIIMDD edge $e$ by traversing the entire LIIMDD. We have $\langle e|e\rangle = |\lambda|^2 \langle v|P^*P|v\rangle = |\lambda|^2 \langle v|v\rangle$, because $P^*P = \mathbb{I}$ for Pauli matrices. Therefore, to this end, it computes the squared norm of $|v\rangle$. Since $|v\rangle = (\text{low}, |\text{low}|v\rangle + (|\text{high}, |\text{high}|v\rangle$, 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 intermediate results in a cache SNORMCACHE: $\text{NODE} \rightarrow \mathbb{R}$ for all recursive calls (Line 7, 8). Therefore, it runs in time $O(m)$ for a diagram with $m$ nodes.
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: \[ s_0 := \text{SquaredNorm}(\text{follow}_0(e)) \]
3: \[ s_1 := \text{SquaredNorm}(\text{follow}_1(e)) \]
4: return $s_0 / (s_0 + s_1)$
5: procedure SquaredNorm(Edge $\lambda \mathcal{P} \circ \mathcal{O}$ with $\lambda \in \mathbb{C}$, $P \in \text{Pauli}^{idx(v)}$)
6: if $\text{idx}(v) = 0$ then return $|\lambda|^2$
7: if $v \notin \text{SNormCache}$ then \[ \triangleright \text{Compute result once for } v \text{ and store in cache:} \]
8: \[ \text{SNormCache}[v] := \text{SquaredNorm}(\text{follow}_0(\mathcal{O}^{-1})) + \text{SquaredNorm}(\text{follow}_1(\mathcal{O}^{-1})) \]
9: return $|\lambda|^2 \cdot \text{SNormCache}[v]$ \[ \triangleright \text{Retrieve result for } e \text{ from cache and multiply with } |\lambda|^2 \]
10: procedure UpdatePostMeas(Edge $e \rightarrow \mathcal{P} \circ \mathcal{O}$, measurement outcome $m \in \{0, 1\}$)
11: if $m = 0$ then
12: \[ e_r := \text{MakeEdge}(\text{follow}_0(e), 0 \cdot \text{follow}_0(e)) \]
13: else
14: \[ e_r := \text{MakeEdge}(0 \cdot \text{follow}_0(e), \text{follow}_1(e)) \]
15: return $1 / \sqrt{\text{SquaredNorm}(e_r)} \cdot e_r$

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 $|e⟩ = |0⟩|e_0⟩ + |1⟩|e_1⟩$ after the top qubit is measured to be $m$, we simply construct an edge $|m⟩|e_m⟩$ 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 $\rightarrow \mathcal{O}$ with $k = \text{idx}(v)$, throw a $p$-biased coin to determine $x_k$, use $\text{follow}_{x_k}(\rightarrow \mathcal{O})$ to go to level $k - 1$ and repeat the process.

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., $L_k \equiv \mathbb{1} \otimes \cdots \otimes L_k \otimes \cdots \otimes \mathbb{1}$.

**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 $A = \lambda P_0 \otimes \cdots \otimes P_1$ to $\lambda P_0 \otimes \cdots \otimes P_{k+1} \otimes Q_k \otimes P_{k-1} \otimes \cdots \otimes P_1$.

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 $e = \longrightarrow \mathcal{O}$, 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 $\begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix}$, we construct a new root node MakeEdge($\alpha \cdot e_0, \beta \cdot e_1$). For the anti-diagonal node $\begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix}$, it is sufficient to note that $\begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix} = X \cdot \begin{bmatrix} 0 & \beta \\ \alpha & 0 \end{bmatrix}$; thus, we can first apply a diagonal gate, and then an $X$ gate, as described above.
Applying a phase gate \((S = [1 \ 0 \ 0 \ 1])\) to qubit with index \(k\) on \(B_{\text{root}}\) is also efficient. Alg. 6 gives a recursive procedure. If \(k < n = \text{idx}(v)\) (top qubit), then note \(S_k B_{\text{root}} v = (S_k B_{\text{root}} S_k^\dagger) S_k v\) where \(S_k B_{\text{root}} S_k^\dagger\) is the new \((O(n))\)-computable root (Pauli)-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 \(\text{MAKEEDGE}\) on Line 6 and 7. To apply \(S_k\) to \(v\) when \(k = n = \text{idx}(v)\), we finally multiply the high edge label with \(i\) on Line 4. Dynamic programming, using table \(\text{SGateCache}\), ensures a linear amount of recursive calls in the number of nodes \(m\). The total runtime is therefore \(O(mn^3)\), as \(\text{MAKEEDGE}\)’s is cubic (see Sec. 4).

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

1: \textbf{procedure} \text{SGate}Edge \(\xrightarrow{A\Delta}v\) with \(A \in \text{Pauli-LIM}, k \in \{1, \ldots, \text{idx}(v)\}\)
2: \textbf{if} \(v \notin \text{SGateCache}\) then \hspace{1em} \triangleright Compute result once for \(v\) and store in cache:
3: \hspace{2em} \textbf{if} \(\text{idx}(v) = k\) then
4: \hspace{3em} \text{SGateCache}[v] := \text{MAKEEDGE}(\text{low}_v, i \cdot \text{high}_v)
5: \hspace{2em} \textbf{else}
6: \hspace{3em} \text{SGateCache}[v] := \text{MAKEEDGE}(\text{SGate}(\text{low}_v, k), \text{SGate}(\text{high}_v, k))
7: \textbf{return} \(S_k A S_k^\dagger \cdot \text{SGateCache}[v]\) \hspace{1em} \triangleright Retrieve result from cache

Applying a Downward Controlled-Pauli gate \(CQ_t\), 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 \(\text{idx}(v) > k\), then since \(CQ_t\) is a Clifford gate, we may push it through the node’s root label, and apply it to the children \(\text{low}(v)\) and \(\text{high}(v)\), similar to the \(S\) gate. Otherwise, if \(\text{idx}(v) = c\), then update \(v\)’s high edge label as \(B \mapsto Q_t B\), and do not recurse. Alg. 7 shows the recursive procedure, which is similar to Alg. 6 and also has \(O(mn^3)\) runtime.

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

1: \textbf{procedure} \text{CPauliGate}Edge \(\xrightarrow{A\Delta}v\) with \(A \in \text{Pauli-LIM}, c, t \in \{1, \ldots, n\}\)
2: \textbf{if} \(v \notin \text{CPauliCache}\) then \hspace{1em} \triangleright Compute result once for \(v\) and store in cache:
3: \hspace{2em} \textbf{if} \(\text{idx}(v) = k\) then
4: \hspace{3em} \text{CPauliCache}[v] := \text{MAKEEDGE}(\text{low}_v, X_t \cdot \text{high}_v)
5: \hspace{2em} \textbf{else}
6: \hspace{3em} \text{CPauliCache}[v] := \text{MAKEEDGE}(\text{CPauliGate}(\text{low}_v, c, t), \text{CPauliGate}(\text{high}_v, c, t))
7: \textbf{return} \(CX_t \cdot A \cdot CX_t^\dagger \cdot \text{CPauliCache}[v]\) \hspace{1em} \triangleright Retrieve result from cache

Sec. 3.3.4 shows that all Clifford gates (including Hadamard and upward CNOT) have runtime \(O(n^4)\) when applied to a stabilizer state represented as a LIMDD. We first show how to apply general gates, in Sec. 3.3.3, as this yields some machinery required for Hadamards (specifically, a pointwise addition operation).

3.3.3 Applying a generic multi-qubit gate to a state

We use a standard approach \([24]\) 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 \(u_{x+y}\) represents a quadrant of the matrix, namely the
submatrix \( u_{xy}(r_2, c_2, \ldots, r_n, c_n) \triangleq u(x, y, r_2, c_2, \ldots, r_n, c_n) \), as follows:

\[
  u = \begin{pmatrix}
    u_{00} & u_{01} \\
    u_{10} & u_{11}
  \end{pmatrix}
\]

(4)

Def. 4 formalizes this idea. Fig. 7 shows a few examples of gates represented as LIMDDs.

**Definition 4 (LIMDDs for gates).** A LIMDD edge \( e = \frac{A}{B} \) can represent a (unitary) \( 2^n \times 2^n \) matrix \( U \) iff \( \text{id}(u) = 2n \). The value of the matrix cell \( U_{r,c} \) is defined as \( \text{FOLLOW}_{r_1, r_2, \ldots, r_n, c_1, \ldots, c_n}(\frac{A}{B}) \) where \( r, c \) are the row and column index, respectively, with binary representation \( r_1, \ldots, r_n \) and \( c_1, \ldots, c_n \). The semantics of a LIMDD edge \( e \) as a matrix is denoted \( [e] \triangleq U \) (as opposed to its semantics \([e]\) as a vector).

The procedure \textbf{ApplyGate} (Alg. 8) applies a gate \( U \) to a state \(|\varphi\rangle\), represented by LIMDDs \( e_U \) and \( e_\varphi \). It outputs a LIMDD representing \( U \otimes |\varphi\rangle \). It works similar to well-known matrix-vector product algorithms for decision diagrams \([24, 27]\), except that we also handle edge weights with LIMs (see Fig. 8 for an illustration). Using the \text{FOLLOW}_{x}(e)\ procedure, we write \(|\varphi\rangle \) and \( U \) as

\[
|\varphi\rangle = |0\rangle |\varphi_0\rangle + |1\rangle |\varphi_1\rangle
\]

(5)

\[
U = |0\rangle \langle 0| \otimes U_{00} + |1\rangle \langle 1| \otimes U_{01} + |1\rangle \langle 0| \otimes U_{10} + |1\rangle \langle 1| \otimes U_{11}
\]

(6)

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 \( |f_{r,c}\rangle = U_{rc} |\varphi_c\rangle \), using four recursive calls to the \textbf{ApplyGate} algorithm. Next, on Lines 7 and 8, the appropriate states are added, using \textbf{ADD} (Alg. 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 \( |e_1\rangle = U_{10} |\varphi_0\rangle + U_{11} |\varphi_1\rangle \). The base case of \textbf{ApplyGate} is the case where \( n = 0 \), which means \( U \) and \( |v\rangle \) are simply scalars, in which case both \( e_U \) and \( e_\varphi \) are edges that point to the leaf.

**Algorithm 8** Applies the gate \(|e_U\rangle\) to the state \(|e_\varphi\rangle\). Here \( e_U \) and \( e_\varphi \) are LIMDD edges. The output is a LIMDD edge \( \psi \) satisfying \(|\psi\rangle = |e_U\rangle |e_\varphi\rangle\).

1: \begin{algorithmic}
2: \textbf{procedure} \textbf{ApplyGate}(Edge \( e_U = \frac{A}{B} \), Edge \( e_\varphi = \frac{\lambda}{Q} \)) with \text{id}(u) = 2 \cdot \text{id}(v))
3: \textbf{if} \( \text{id}(v) = 0 \) \textbf{then return} \( \frac{\lambda \gamma}{Q} \)
4: \textbf{if} \((P', u, Q', v) \notin \text{APPLY-CACHE}) \textbf{then}
5: \textbf{for} \( r, c \in \{0, 1\} \) \textbf{do}
6: \textbf{if} \( f_{r,c} := \text{APPLY-CACHE}((P'|Q|)) \) \textbf{then}
7: \textbf{end if}
8: \textbf{end for}
9: \textbf{return} \( \frac{\lambda \gamma \cdot e_\psi}{Q} \)
\end{algorithmic}

Caching in \textbf{ApplyGate}. A straightforward way to implement dynamic programming would be to simply store all results of \textbf{ApplyGate} in the cache, i.e., when \textbf{ApplyGate}(\( \frac{A}{B} \), \( \frac{\lambda}{Q} \)) is

Accepted in \textit{Quantum} 2023-07-17, click title to verify. Published under CC-BY 4.0.
called, store an entry with key \((P, u, Q, v)\) in the cache. This would allow us to retrieve the result the next time \textsc{ApplyGate} is called with the same parameters. However, we can do much better, in such a way that we can retrieve the result from the cache also when the procedure is called with parameters \textsc{ApplyGate}\((\frac{\lambda P}{A}, \frac{\gamma Q}{B})\) satisfying \(|\lambda P\rangle = |\frac{\lambda P}{A}\rangle\) and \(|\gamma Q\rangle = |\frac{\gamma Q}{B}\rangle\). 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 \textsc{RootLabel}, which returns a canonically chosen LIM, i.e., it holds that \(\textsc{RootLabel}(\frac{\lambda}{A}) = \textsc{RootLabel}(\frac{\gamma}{B})\) whenever \(A \mid v \rangle = B \mid v \rangle\). A specific choice for \textsc{RootLabel} is the lexicographic minimum of all possible root labels. In Alg. 17, we give an \(O(n^2)\)-time algorithm for computing the lexicographically minimal root label, following the same strategy as the \textsc{MakeEdge} procedure in Sec. 4.2. As a last optimization, we opt to not store the scalars \(\lambda, \gamma\) in the cache (they are “factored out”), so that we can retrieve this result also when \textsc{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 \textsc{Add} (Alg. 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 \(\frac{A + B}{A + B}\).

**Algorithm 9** Given two \(n\)-LIMDD edges \(e, f\), constructs a new LIMDD edge \(a\) with \(\{|a\rangle = |e\rangle + |f\rangle\}.

1: \textbf{procedure} \textsc{Add}(Edge \(e\) = \(\frac{A}{A}\), Edge \(f\) = \(\frac{B}{B}\) with \(\text{idx}(v) = \text{idx}(w)\))
2: \textbf{if} \(\text{idx}(v) = 0\) \textbf{then return} \(\frac{A + B}{A + B}\) \triangleright A, B \in \mathbb{C}
3: \textbf{if} \(v \not\in w\) \textbf{then return} \textsc{Add}(\(\frac{B}{B}\), \(\frac{A}{A}\)) \triangleright \text{Normalize for cache lookup}
4: \(C := \textsc{RootLabel}(\frac{A^{-1}B}{A^{-1}B})\)
5: \textbf{if} \((v, C, w) \not\in \textsc{Add-Cache}\) \triangleright \text{Compute result for the first time:
6: \hspace{1em} \textbf{Edge} \(a_0 := \textsc{Add-Cache}(\rightarrow, \text{follow}_0(\frac{C}{C}))\)
7: \hspace{1em} \textbf{Edge} \(a_1 := \textsc{Add-Cache}(\rightarrow, \text{follow}_1(\frac{C}{C}))\)
8: \hspace{1em} \textbf{Add-Cache}[(v, C, w)] := \textsc{MakeEdge}(a_0, a_1)\)
9: \textbf{return} \(A \cdot \textsc{Add-Cache}[(v, C, w)]\) \triangleright \text{Retrieve from cache}

**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 \textsc{Add}(\(\frac{A}{A}, \frac{B}{B}\)). However, we can do much better; namely, we remark that we are looking to construct the state \(A \mid v \rangle + B \mid w \rangle\), and that this is equal to \(A \cdot |v\rangle + A^{-1}B \mid w \rangle\). 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 \(C = \textsc{RootLabel}(\frac{A^{-1}B}{A^{-1}B})\) (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$(\stackrel{D}{\text{2}}, \stackrel{E}{0})$ 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 $O(n^32^n)$ (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 [52, Table 2]. The reason for this is that addition may remove any common factors, as illustrated in Fig. 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. 3.3.4.

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 can be applied to a stabilizer state in polynomial time. Together with the results of Sec. 3.3.2, this shows that all Clifford gates can be applied to stabilizer states in polynomial time. The key ingredient is Th. 7, which describes situations in which the ADD procedure looks up the same tuples in the cache in both its recursive calls (modulo $\pm1$). Th. 5 gives the final result.

**Theorem 5.** Any Clifford gate $(H, S, \text{CNOT})$ can be applied in $O(n^4)$ 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 $\stackrel{A}{\text{1}}\text{X}$. By Th. 1, this LIMDD is a $(\text{Pauli})$-Tower-LIMDD with $m = n$ nodes apart from the leaf.

Sec. 3.3.2 shows that any $S$-gate can be applied in time $O(n^3m)$, so we get $O(n^4)$. Th. 6 shows that any Hadamard gate can be applied on any qubit in time $O(n^4)$.

Sec. 3.3.3 shows that any downward CNOT-gate can be applied in time $O(n^3m)$, so in this case $O(n^4)$. By applying Hadamard to the target and control qubits, before and after the downward CNOT, we obtain an upward CNOT, i.e., $CX^t = (H \otimes H)CX^t(H \otimes H)$, still in time $O(n^4)$.

To apply a Hadamard gate $(H = \frac{1}{\sqrt{2}}[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}])$ 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 (Alg. 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. 3.3.2. Alg. 10 shows the complete algorithm.

**Theorem 6.** Let $e$ be an $n$-qubit $(\text{Pauli})$-Tower-LIMDD. HGate$(e,k)$ of Alg. 10 takes $O(n^4)$ time.

**Proof.** 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. By applying induction over the qubits $1, \ldots, k$, using
Th. 7, it is easy to see that at each level, the cache in Alg. 9 is consulted at Line 5 with a tuple \((v, k)\) where \(k\) is a count. This tells us that \(ADD\) performs at most \(2k\) recursive calls. Each recursive call to \(ADD\) may invoke the \(MAKEEDGE\) procedure, which runs in time \(O(n^3)\), yielding a total worst-case running time of \(O(n^4)\), when \(k = \Omega(n)\).

**Theorem 7.** If Alg. 9 is called on two edges pointing to the same (Pauli)-Tower-LIMDD node \(v\) with \(\text{low}(v) = \text{high}(v)\), then the recursive \(ADD\) calls at Line 6, 7 both lookup the same LIM in cache up to a factor \(\pm 1\).

**Proof.** Assume the algorithm is at Line 6. Let \(\square \rightarrow Q \rightarrow Q\) be a node on which the algorithm was called. Let \(C = P_n \otimes P\) be the \(n\) qubit Pauli-LIM computed at Line 4 with \(P_n \in \text{Pauli}\) and \(P\) an \(n-1\) qubit Pauli-LIM. At Line 6 and 7, \(ADD\) makes two recursive calls computing \(a_x\) for \(x \in \{0, 1\}\), as listed in the header of the below table. The \(\text{follow}(x)\) semantics yield four cases for \(x = 1\) recursive \(ADD\) calls, depending on \(x\) and \(P_n\). The following table shows the tuples computed for cache normalization at Line 4 in the recursive call, ignoring the \(\text{RootLabel}\) function for now. E.g., if \(\sim\) denotes cache normalization, then \(\gamma_r \rightarrow_R \circ R \rightarrow_Q \circ Q \rightarrow (v, \pm Q, w)\) since \(QR = \pm RQ\) for \(P, Q \in \text{Pauli-LIM}\):

<table>
<thead>
<tr>
<th>(a_0 := \text{ADD}(\text{follow}(0 \rightarrow \square), \text{follow}(Q \rightarrow \square)))</th>
<th>(a_1 := \text{ADD}(\text{follow}(1 \rightarrow \square), \text{follow}(Q \rightarrow \square)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_n = I):</td>
<td>(P_n = X):</td>
</tr>
<tr>
<td>(\square \rightarrow P \rightarrow \square \sim (w, P, w))</td>
<td>(\square \rightarrow PQ \rightarrow \square \sim (w, PQ, w))</td>
</tr>
<tr>
<td>(\square \rightarrow PQ \rightarrow \square \sim (w, PQ, w))</td>
<td>(\square \rightarrow iPQ \rightarrow \square \sim (w, iPQ, w))</td>
</tr>
</tbody>
</table>

In all four cases, the cache-normalized LIMs computed in both recursive calls are equivalent up to a factor \(\pm 1\). Finally, our \(\text{RootLabel}\) function from Sec. 4.2.1, which selects the lexicographic smallest label, satisfies \(\text{RootLabel}(Q \rightarrow \square) = -\text{RootLabel}(-Q \rightarrow \square)\) for any PauliLIM \(Q\). This completes the proof.

Since stabilizer states are closed under Clifford gates, (Pauli)-Tower-LIMDDs should also be closed under the respective LIMDD manipulation operations. We show this in Lemma 4 (App. C).

### 3.4 Comparing LIMDD-based simulation with other methods

Prop. 1 shows exponential advantages of (Pauli)-LIMDDs over three state-of-the-art classical quantum circuit simulators: those based on QMDDs and MPS [36, 37], 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 Th. 8.

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 [38] and QMDD-based [28] simulation—proceeds by representing a state $|\varphi_t\rangle$ at time step $t$ as a LIMDD $\varphi_t$. It then applies the gate $U_t \in G$ in the circuit corresponding to this time step to obtain a LIMDD $\varphi_{t+1}$ 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. 3.1).

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 [53] and de Bakker & Meertens [54]). 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.

### 3.4.1 LIMDD is exponentially faster than QMDD-based simulation

As state set $S$, we select the stabilizer states and for $G$ the Clifford gates. Th. 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 Def. 3). Sec. 3.3.2 shows that LIMDD can execute all Clifford gates on stabilizer states in time $O(n^4)$.

On the other hand, Th. 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 $\text{QSim}^{\text{LIMDD}} = \Omega(2^n \cdot \text{QSim}^{\text{QMDD}})$ (Prop. 1 Item 3).

For the other direction, we now show that LIMDDs are at most a factor $O(n^3)$ 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. 3.1). The LIMDD additionally uses $O(n)$ memory per node to store two Pauli LIMs; thus, the total memory usage is at most a factor $O(n)$ worse than QMDDs for any given state. The ApplyGate and Add algorithms introduced in Sec. 3.3.3 are very similar to the ones used for QMDDs in [1, 24]. 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 $O(n^3)$ instead of $O(1)$. 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 $O(n^3)$.

Finally, Corollary 3 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 [55, Th 2.4.1]. Together with the universal simulation discussed above, this proves that the above also holds for a simulator based on the combination QMDD $\cup$ Stab (Prop. 1 Item 5).

### 3.4.2 LIMDD is exponentially faster than MPS

In Sec. 3.4.1, we saw that LIMDD can simulate the cluster state in polynomial time. On the other hand, Th. 4 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. 1 Item 2.

### 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 accepted).
believed to be true). This method decomposes a given quantum circuit into a circuit consisting only of Clifford gates and the $T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\pi i/4} \end{bmatrix}$ gate, as explained in Sec. 2.

The circuit family, given my McClung [56], maps the input state $|0\rangle^\otimes n$ to the $n$-qubit $W_n$ state $|W_n\rangle$, which is the equal superposition over computational-basis states with Hamming weight 1,

$$|W_n\rangle = \frac{1}{\sqrt{2^n}} \left( |100\ldots00\rangle + |010\ldots00\rangle + \cdots + |000\ldots01\rangle \right)$$

Arunachalam et al. showed that, assuming ETH, any circuit which deterministically produces the $|W_n\rangle$ state in this way requires $\Omega(n) \cdot T$ gates [57]. 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 [58, 59]).

Th. 8 now shows that the exponential separation between simulation with LIMDD and Clifford + $T$, i.e., that $QSim_{Clifford + T} = \Omega(2^n \cdot QSim_{^{LIMDD}T})$ (Prop. 1 Item 1). App. F gives its proof.

**Theorem 8.** There exists a circuit family $C_n$ such that $C_n \cdot |0\rangle^\otimes n = |W_n\rangle$, 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 [60–62]. 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. [14] 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 $O(n^2)$-size LIMDD can be obtained via a construction by Bryant [19], 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 App. G for details.

## 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 [20, 27], by facilitating efficient manipulation operations through dynamic programming efficiently, as discussed in Sec. 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 $\approx_{\text{Pauli}}$ 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 Fig. 10. However, by imposing a small number of constraints on the diagram, listed in Def. 5 and visualized in Fig. 11, we ensure that every quantum state is represented by a unique ‘reduced’ Pauli-LIMDD. We present a MakeEdge algorithm (Alg. 11 in Sec. 4.2) that computes a canonical node assuming its children are already canonical. The algorithms for quantum circuit simulation in Sec. 3.3 ensure that all intermediate LIMDDs are reduced by creating nodes exclusively through this subroutine.
Figure 10: Four different Pauli-LIMDDs representing the Bell state \( \sqrt{1/2}(|00\rangle + |11\rangle) \). 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 \( \implies X \) 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.

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 algorithms always construct the diagram in the backtrack of the recursion using a typical ‘MakeNode’ procedure for constructing canonical nodes [24], like in Sec. 3.3.) For instance, a QMDD node \( \bigotimes_{\alpha}^I \bigotimes_{\beta}^X \) with \( \alpha, \beta \in \mathbb{C} \setminus \{0\} \) can be reduced into a canonical node by dividing out a common factor \( \alpha \) and placing it on the root edge. Assuming that \( v, w \) are canonical, the resulting node \( \bigotimes_{\alpha}^1 \bigotimes_{\beta/\alpha}^X \) 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 [27] 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 \( \beta/\alpha \) in the example above), as the Pauli-LIM group is more complicated than the group of complex numbers (scalars).

Def. 5 gives reduction rules for LIMDDs and Fig. 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 Fig. 10. The low precedence rule (3) makes this choice deterministic, but only in case \( \text{low}(v) \neq \text{high}(v) \). Next, the zero edges (2) rule handles the case when \( \alpha \) or \( \beta \) 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 Def. 2). The rule forces \( \text{low}(v) = \text{high}(v) \) 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. 4.2.

Definition 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 \( \text{low}(v) = \text{low}(w), \text{high}(v) = \text{high}(w), \text{label}(\text{low}(v)) = \text{label}(\text{low}(w)), \text{label}(\text{high}(v)) = \text{label}(\text{high}(w)) \).

2. **(Zero) edge:** For any edge \( (v, w) \in \text{high} \cup \text{low} \), if \( \text{label}(v, w) = 0 \), then both edges outgoing from \( v \) point to the same node, i.e., \( \text{high}(v) = \text{low}(v) = w \).

3. **Low precedence:** Each node \( v \) has \( \text{low}(v) \preceq \text{high}(v) \), where \( \preceq \) is a total order on nodes.

4. **Low factoring:** The label on every low edge to a node \( v \) is the identity \( I^{\otimes \text{idx}(v)} \).
(1) Merge $u, v$ into $v$
(2) Zero edges
(3) Low precedence with $u \leq w$
(4) Low factoring
(5) High determinism

\[ H = \text{HighLabel}(v) \]

Figure 11: Illustrations of the reduction rules from Def. 5 applied at level $k+1$ (i.e., $k+1 = \text{idx}(v) = \text{idx}(v')$).

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).

5. **High determinism:** The label on the high edge of any node $v$ is $B_{\text{high}} = \text{HighLabel}(v)$, where $\text{HighLabel}$ is a function that takes as input a semi-reduced $n$-PAULI-LIMDD node $v$, and outputs an $(n-1)$-PAULI-LIM $B_{\text{high}}$ satisfying $|v| \simeq_{\text{PAULI}} |0\rangle \langle 0| + |1\rangle \langle 1| B_{\text{high}} |\text{high}(v)|$.

Moreover, for any other semi-reduced node $w$ with $|v| \simeq_{\text{PAULI}} |w|$, it satisfies $\text{HighLabel}(w) = B_{\text{high}}$. In other words, the function $\text{HighLabel}$ 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 $|\varphi\rangle \neq_{\text{PAULI}} |\psi\rangle$, where $A, B \in \text{PAULI}$. Assume we already have canonical LIMDDs for $\varphi$ and $\psi$ (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 $\bigotimes^{n-1} A \rightarrow \bigotimes$ or $\bigotimes^{n-1} B \rightarrow \bigotimes$, as these are related by the isomorphism $X \otimes I$. 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 $\bigotimes^{n-1} A \rightarrow \bigotimes^{n-1} B \rightarrow \bigotimes$ (with root edge $I \otimes A$ as in Fig. 11 (4)). Otherwise, if $\psi \prec \varphi$, we obtain node $\bigotimes^{n-1} A \rightarrow \bigotimes^{n-1} B \rightarrow \bigotimes$ with incoming edge $X \otimes B$. Finally, since there might be other LIMs $B_{\text{high}}$ not equal to $B^{-1}A$ that yield the same state, the high determinism rule is finally needed to obtain a canonical node $\bigotimes^{n-1} A \rightarrow \bigotimes^{n-1} B_{\text{high}} \rightarrow \bigotimes$ as shown in Fig. 12. This last step turns a semi-reduced node into a (fully) reduced node. Sec. 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 $\text{low}(v) = \text{high}(v) = \varphi$ in this case. There is a choice between representing this state as either $\bigotimes^{n-1} A \rightarrow \bigotimes$ or $\bigotimes^{n-1} B \rightarrow \bigotimes$, which denote the states $|0\rangle \otimes A|\varphi\rangle$ and $|1\rangle \otimes A|\varphi\rangle$, as these are related by the isomorphism $X \otimes I$. The low factoring rule requires that the low edge label is $I$, yielding a node of the form $\bigotimes^{n-1} A \rightarrow \bigotimes$ 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 1. This is not a problem, since the 0-vector is not a quantum state.

The rules in Def. 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. 3.2). 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 $v_0 \leq v_1$ for every node. Hence, in this case, reduced $G$-LIMDD cannot represent a state $|0\rangle \langle v_0| + |1\rangle \langle v_1|$ when $v_1 < v_0$. However, it is not difficult to formulate rules to support these groups $G$; for instance, when $G = \{I\}$, we recover the QMDD and may use its reduction rules [28].

Nodes and edges in a reduced LIMDD need not represent normalized quantum states, just like in (unreduced) LIMDDs as explained in Sec. 3.1. Consider, e.g., node $\ell_2$ in Fig. 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 \cite{18, 19, 48} 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 \footnote{\textsuperscript{\textdagger}}. However, if needed Def. \textdaggerleft{2\textdaggerright} could be adapted and a deletion rule could be added to Def. \textdaggerleft{5\textdaggerright}.

We now give a proof of Th. \textsuperscript{9}, which states that reduced LIMDDs are canonical.

**Theorem \textsuperscript{9} (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 $\lambda$. 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 $id(v) > 0$, 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 $\overset{\lambda A}{\rightarrow} 1$.

**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 $|\varphi_0\rangle, |\varphi_1\rangle \neq 0$.

**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 $|\varphi\rangle$: $\overset{A}{\otimes I} \otimes \overset{\lambda A}{\rightarrow} \otimes \overset{\lambda A}{\rightarrow}$ as illustrated in Fig. 12 (left). In case $|\varphi_1\rangle \neq 0$, we do the same for root node $v_R$ as in case $|\varphi_1\rangle \neq 0$, we do the same for root $r = A|w\rangle$, but switch the high and the low edge by instead a root edge $e_r = X \otimes A$ (similar to Fig. 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 $(\emptyset, w, 0, 0)$, where $w$ is canonical because of the induction hypothesis. Also in both cases, we also have $|\varphi\rangle = |v\rangle$ because either $|\varphi\rangle = \I \otimes A|w\rangle$ or $|\varphi\rangle = X \otimes A|w\rangle$.

**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 $w$ 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 $v_L \not\simeq v_R$, we construct the following reduced Pauli-LIMDD for $|\varphi\rangle$:

- the root node is $\bigotimes A \circ E \circ B$, where $E$ is the LIM computed by $\text{HighLabel}(\bigotimes A^{-1} B)$.

Otherwise, if $v_R \not\simeq v_L$, then we construct the following reduced Pauli-LIMDD for $|\varphi\rangle$: the root node is $\bigotimes A \circ E \circ B$, where $F = \text{HighLabel}(\bigotimes A^{-1} B)$. It is straightforward to check that, in both cases, this Pauli-LIMDD is reduced. Moreover, $|v\rangle$ isomorphic to $|\varphi\rangle$ as illustrated in Fig. 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,

\[
\bigotimes A \circ E \circ B
\]

(7)

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 $P = P_{\text{top}} \otimes P_{\text{rest}} \neq 0$ where $P_{\text{top}}$ is a single-qubit Pauli matrix and $P_{\text{rest}}$ an $(n-1)$-qubit Pauli LIM. Expanding the semantics of $v_L$ and $v_M$, we obtain,

\[
\lambda P_{\text{top}} \otimes P_{\text{rest}} ([0] \otimes A_L |v^0_L\rangle + [1] \otimes B_L |v^1_L\rangle) = [0] \otimes A_M |v^0_M\rangle + [1] \otimes B_M |v^1_M\rangle.
\]

(8)

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

Case $P_{\text{top}} = I, Z$. If $P_{\text{top}} = [1 \ 0; 0 \ 1]$ for $z \in \{ 1, -1 \}$, then Eq. 8 gives:

\[
\lambda P_{\text{test}} A_L |v^0_L\rangle = A_M |v^0_M\rangle \quad \text{and} \quad \lambda P_{\text{test}} B_L |v^1_L\rangle = B_M |v^1_M\rangle
\]

(9)

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

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

$B_L = 0 \text{ or } B_M = 0$: In case $B_L = 0$, we see from Eq. 9 that $0 = B_M |v^1_M\rangle$. Since the state vector $|v^1_M\rangle \neq 0$ 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 Eq. 9 yields $z \lambda P_{\text{rest}} B_L |v^1_L\rangle = 0$. We have $z \lambda \neq 0$, $P_{\text{rest}} \neq 0$ by definition, and we observed $|v^1_L\rangle \neq 0$ 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 $P_{\text{top}} = [0 \ 1; z \ 0]$ for $z \in \{ 1, i \}$, then Eq. 8 gives:

\[
\lambda z P_{\text{test}} A_L |v^0_L\rangle = B_M |v^1_M\rangle \quad \text{and} \quad \lambda z^* P_{\text{test}} B_L |v^1_L\rangle = A_M |v^0_M\rangle.
\]

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

$B_L \neq 0 \text{ and } B_M \neq 0$: we find $|v^0_L\rangle \simeq |v^0_M\rangle$ and $|v^1_L\rangle \simeq |v^0_M\rangle$, so by the induction hypothesis, $v^0_L = v^0_M$ and $v^1_L = v^1_M$. By low precedence, it must be that $v^1_L = v^0_M = v^1_M$. Now use high determinism to infer that $B_L = B_M$ as in the $P_{\text{top}} = I, Z$ case.
In order to choose the canonical high edge label of node \( v \), the MAKEEDGE algorithm calls GETLABELS (Line 10 of Alg. 11). The function GETLABELS returns a uniquely chosen LIM \( B_{\text{high}} \) 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 App. D (Alg. 12). First, we characterize the eligible high-edge labels. That is, given a semi-reduced node \( \otimes^{1\leq i < j \leq n} \langle i | A_j | j \rangle \), we characterize all \( C \) such that the node \( \otimes^{1\leq i < j \leq n} \langle i | C_j \cdot A_j | j \rangle \) is isomorphic to \( \otimes^{1\leq i < j \leq n} \langle i | A_j | j \rangle \). Our characterization shows that, modulo some complex factor, the eligible labels \( C \) are of the form

\[
C \propto \lambda \cdot \hat{A} \cdot \hat{g}_0 \cdot \hat{g}_1, \quad \text{for } \lambda \in \text{Stab}(|v_0\rangle), \ g_0 \in \text{Stab}(|v_0\rangle), \ g_1 \in \text{Stab}(|v_1\rangle)
\]

(10)

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. (10) shows that this set has a polynomial-size description by storing only the generators of the stabilizer subgroups.
We mention related work on classical simulation formalisms and decision diagrams other than 
QMDD.

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 Def. 5), there is a unique LIMDD representing a quantum state up to phase and local Pauli operators.

5 Related work

We mention related work on classical simulation formalisms and decision diagrams other than QMDD.
The Affine Algebraic Decision Diagram, introduced by Tafertshofer and Pedam [63], and by Sanner and McAllister [26], is akin to a QMDD except that its edges are labeled with a pair of real numbers \((a, b)\), so that an edge \((a, b)\) represents the state vector \(a|v\rangle + b|+\rangle^⊗n\) (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) [64, 65] extend BDDs with insights from visibly pushdown automata [66]. An extension of CFLOBDD to the complex domain [67] shows good performance for various simulation of quantum computing benchmarks. Sentential Decision Diagrams [68] generalize BDDs by replacing their total variable order with a variable tree (vtree). Although Kisa et al. [69] 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) [70] improves on the SDD by merging isomorphic vtree nodes. We remark that CFLOBDDs are similar to VS-SDD with a balanced vtree.

Günter and Drechsler introduced a BDD variant [71] 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 \(A \in \mathbb{F}^{2n \times n}_2\). If the root node represents the function \(r: \mathbb{B}^n \rightarrow \mathbb{B}\), then the diagram represents the function \(f(\bar{x}) = r(A \cdot \bar{x})\). (This concept 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 \(+, ×\) 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 [33, 45].

6 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 \(G = \text{Pauli}, G = \langle Z \rangle\) and \(G = \langle X \rangle\), and found that any choice suffices for an exponential advantage over QMDDs; notably, the choice \(G = \text{Pauli}\) 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 [72–74] and tensor networks [29, 75]. It would also be interesting to compare LIMDDs to graphical calculi such as the ZX calculus [76], following similar work for QMDDs [77].

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 [55, 78], making it interesting to investigate variants of LIMDDs with a flexible ordering, for example taking inspiration from the Sentential Decision Diagram [68?].

7 Acknowledgements

We thank the anonymous reviewers at Quantum for their valuable suggestions, which helped to greatly improve the presentation of the manuscript. We thank Dan Browne for help with establishing stabilizer ranks. We thank Marie Anastacio, Jonas Helsen, Yash Patel and Matthijs Rijlaarsdam for their feedback on early versions of the manuscript. We thank Patrick Emonts and Adrià Pérez-Salinas for discussions on MPS, and Kenneth Goodenough for useful discussions in general. The second author acknowledges the QIA project (funded by European Union’s Horizon 2020, Grant Agreement No. 820445). The third and fourth author are funded by the Netherlands Organization for Scientific Research (NWO/OCW), as part of the Quantum Software Consortium program (Project No. 024.003.037/3368). The last author is funded by the research program VENI with project number 639.021.649 of the Netherlands Organization for Scientific Research (NWO).

References


[35] Eric Chitambar, Debbie Leung, Laura Mančinska, Maris Ozols, and Andreas Winter. “Everything you always wanted to know about LOCC (but were afraid to ask)”. Communications in Mathematical Physics 328, 303–326 (2014).


A Linear-algebra algorithms for Pauli operators

In Sec. 2, we defined the stabilizer group for an n-qubit state |φ⟩ as the group of Pauli operators \( A \in \text{Pauli} \), which stabilize |φ⟩, i.e., \( A|φ⟩ = |φ⟩ \). 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 \( \in \{±1, ±i\} \)) be written as \((X^{x_1}Z^{z_1}) \otimes \cdots \otimes (X^{x_n}Z^{z_n})\) for bits \( x_j, z_j, 1 \leq j \leq n \). We can therefore write an n-qubit Pauli string \( P \) as a length-2\( n \) binary vector as follows [7],
\[
\begin{pmatrix}
X \text{ block} \\
Z \text{ block}
\end{pmatrix}
\begin{pmatrix}
x_n, x_{n-1}, \ldots, x_1 \\
z_n, z_{n-1}, \ldots, z_1
\end{pmatrix},
\]
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 \( Z \otimes I < Z \otimes X \) 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.
\[
\begin{pmatrix}
X \otimes X \otimes X \\
I \otimes Z \otimes Y
\end{pmatrix}
\sim
\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1
\end{pmatrix}.
\]
Furthermore, if \( P, Q \) are Pauli strings corresponding to binary vectors \((x^P, z^P)\) and \((x^Q, z^Q)\), then
\[
P \cdot Q \propto \bigotimes_{j=1}^{n} (X^{x_j^P}Z^{z_j^P}) (X^{x_j^Q}Z^{z_j^Q}) = \bigotimes_{j=1}^{n} (X^{x_j^P \oplus x_j^Q}Z^{z_j^P \oplus z_j^Q})
\]
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., \( \mathbb{F}_2^n \)) 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 \( G = \{g_1, \ldots, g_k\} \) are length-2\( n \) 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 [79] 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 [80] for computing the intersection of two subspaces of a vector space, in time \( O(n^3) \).
Division remainder: given a vector \( h \) (or Pauli string \( h \)), determine \( h_{\text{rem}} := \min_{g \in \langle G \rangle} \{ g \odot h \} \) (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. \( \text{if } h_j = 1 \) and \( G \) has a row \( g_i \) with its pivot at position \( j \) then \( h := h \odot g_i \)

The resulting \( h \) is \( h_{\text{rem}} \). 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 \( \theta \), e.g.:

\[
\begin{pmatrix}
3X & \otimes & X & \otimes & X \\
-\frac{i}{2} I & \otimes & Z & \otimes & Y
\end{pmatrix} \sim
\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 3 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & \frac{3\pi}{2}
\end{pmatrix}
\]

where we used \( 3 = 3 \cdot e^{i0} \) and \( -\frac{i}{2} = \frac{1}{2} \cdot e^{3\pi/2} \). This extended check vector also easily allows a total ordering, namely, we simply use the ordering on real numbers for \( r \) and \( \theta \). For example, \((1, 1, [0, 0, 2, \frac{1}{2}]) < (1, 1, [0, 3, 0]) \). Let us stress that the factor encoding \((r, \theta)\) is less significant than the Pauli string encoding \((x_1, \ldots, x_n)\). 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 Pauli-LIMs input. (When the input consist of pairwise commuting Pauli operators, such as stabilizer subgroups [7], 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 \( \pm \) 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. 4.2.1 and Sec. D.1.

B Proof that cluster states and coset states need exponentially-large QMDDs

In this appendix, we show that QMDDs which represent both clusters states, and coset states, are exponentially large in the worst case (respectively, Th. 2 and Corollary 11). On the other hand, in App. C, we will show that these states can be represented using only \( O(n) \) nodes by \( \langle X \rangle \)-LIMDDs, showing that they are exponentially more succinct than QMDDs. We first fix notation and definitions, after which we prove the theorem using two lemmas.

Let \( G \) be an undirected graph with vertices \( V_G = \{v_1, \ldots, v_n\} \) and edge set \( E_G \subseteq V_G \times V_G \). For a subset of vertices \( S \subseteq V_G \), the \( S \)-induced subgraph of \( G \) has vertices \( S \) and edge set \((S \times S) \cap E \). Given \( G \), its graph state \( |G\rangle \) is expressed as

\[
|G\rangle = \sum_{\vec{x} \in \{0,1\}^n} (-1)^{f_G(\vec{x})} |\vec{x}\rangle
\]  

(11)

where \( f_G(\vec{x}) \) is the number of edges in the \( S \)-induced subgraph of \( G \).

For a function \( f : \{0,1\}^n \to \mathbb{C} \) and bit string \( \vec{a} = a_1 \cdots a_k \in \{0,1\}^k \), we denote by \( f_{\vec{a}} \) the subfunction of \( f \) restricted to \( \vec{a} \):

\[
f_{\vec{a}}(x_{k+1}, \ldots, x_n) := f(a_1, \ldots, a_k, x_{k+1}, \ldots, x_n)
\]  

(12)

Accepted in Quantum 2023-07-17, click title to verify. Published under CC-BY 4.0.
We also say that \( f_\bar{a} \) is a subfunction of \( f \) of order \(|\bar{a}| = k\).

We will also need the notions of boundary and strong matching.

**Definition 6** (Boundary). For a set \( S \subseteq V_G \) of vertices in \( G \), the boundary of \( S \) is the set of vertices in \( S \) adjacent to a vertex outside of \( S \).

**Definition 7** (Strong Matching). Let \( G = (V, E) \) be an undirected graph. A strong matching is a subset of edges \( M \subseteq E \) that do not share any vertices (i.e., it is a matching) and no two edges of \( M \) are incident to the same edge of \( G \), i.e., an edge in \( E \setminus M \). Alternatively, a strong matching is a matching \( M \) s.t. \( G[V(M)] = M \). We say that \( M \) is an \((S,T)\)-strong matching for two sets of vertices \( S, T \subseteq V \) if \( M \subseteq S \times T \). For a strong matching \( M \) and a vertex \( v \in V(M) \), we let \( M(v) \) denote the unique vertex to which \( v \) is matched by \( M \).

Using these definitions and notation, we prove Th. 2.

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

**Proof.** Let \( G = \text{lattice}(n,n) \) be the undirected graph of the \( n \times n \) lattice, with vertex set \( V = \{v_1, \ldots, v_n^2\} \). Let \( \sigma = v_1 v_2 \cdots v_n^2 \) be a variable order, and let \( S = \{v_1, v_2, \ldots, v_{\lfloor n^2/2 \rfloor}\} \subseteq V \) be the first \( \frac{1}{2}n^2 \) vertices in this order.

The proof proceeds broadly as follows. First, in Lemma 1, we show that any \((S, \overline{S})\)-strong matching \( M \) effects \( 2^{|M|} \) different subfunctions of \( f_G \). Second, Lemma 2 shows that the lattice contains a large \((S, \overline{S})\)-strong matching for any choice of \( S \). Put together, this will prove the lower bound on the number of QMDD nodes as in Th. 2 by the fact that a QMDD for the cluster state \( G \) has a node per unique subfunction of the function \( f_G \). Fig. 13 illustrates this setup for the \( 5 \times 5 \) lattice.

**Lemma 1.** Let \( M \) be a non-empty \((S, \overline{S})\)-strong matching for the vertex set \( S \) chosen above. If \( \sigma = v_1 v_2 \cdots v_n^2 \) is a variable order where all vertices in \( S \) appear before all vertices in \( \overline{S} \), then \( f_G(x_1, \ldots, x_n^2) \) has \( 2^{|M|} \) different subfunctions of order \(|S|\).

**Proof.** Let \( S_M := S \cap V(M) \) and \( \overline{S}_M := \overline{S} \cap M \) be the sets of vertices that are involved in the strong matching. Write \( \chi(x_1, \ldots, x_n) \) for the indicator function for vertices: \( \chi(x_1, \ldots, x_n) := \{v_i \mid x_i = 1, i \in [n]\} \). Choose two different subsets \( A, B \subseteq S_M \) and let \( \bar{a} = \chi^{-1}(A) \) and \( \bar{b} = \chi^{-1}(B) \) be the corresponding length-\(|S|\) bit strings. These two strings induce the two subfunctions \( f_{G,\bar{a}} \) and \( f_{G,\bar{b}} \). We will show that these subfunctions differ in at least one point.

First, if \( f_{G,\bar{a}}(0, \ldots, 0) \neq f_{G,\bar{b}}(0, \ldots, 0) \), then we are done. Otherwise, take a vertex \( s \in A \oplus B \) and say w.l.o.g. that \( s \in A \setminus B \). Let \( t = M(s) \) be its partner in the strong matching. Then we have, \( |E[A \cup \{t\}]| = |E[A]| + 1 \) but \( |E[B \cup \{t\}]| = |E[B]| \). Therefore we have

\[
\begin{align*}
f_{G,\bar{a}}(0, \ldots, 0, x_t = 0, 0, \ldots, 0) & \neq f_{G,\bar{a}}(0, \ldots, 0, x_t = 1, 0, \ldots, 0) \\
f_{G,\bar{b}}(0, \ldots, 0, x_t = 0, 0, \ldots, 0) & = f_{G,\bar{b}}(0, \ldots, 0, x_t = 1, 0, \ldots, 0)
\end{align*}
\]

We see that each subset of \( S_M \) corresponds to a different subfunction of \( f_G \). Since there are \( 2^{|M|} \) subsets of \( M \), \( f_G \) has at least that many subfunctions.

We now show that the \( n \times n \) lattice contains a large enough strong matching.

**Lemma 2.** Let \( S = \{v_1, \ldots, v_{\lfloor n^2/2 \rfloor}\} \) be a set of \( \frac{1}{2}n^2 \) vertices of the \( n \times n \) lattice, as above. Then the graph contains a \((S, \overline{S})\)-strong matching of size at least \( \lfloor \frac{1}{12}n \rfloor \).

**Proof.** Consider the boundary \( B_S \) of \( S \). This set contains at least \( n/3 \) vertices, by Theorem 11 in [49]. Each vertex of the boundary of \( S \) has degree at most 4. It follows that there is a set of \( \lfloor \frac{1}{12} |B_S| \rfloor \) vertices which share no neighbors. In particular, there is a set of \( \lfloor \frac{1}{12} |B_S| \rfloor \geq \lfloor \frac{1}{12} n \rfloor \) vertices in \( B_S \) which share no neighbors in \( \overline{S} \).
Put together, every choice of half the vertices in the lattice yields a set with a boundary of at least \(\frac{n}{3}\) nodes, which yields a strong matching of at least \(\left\lfloor \frac{1}{12}n \right\rfloor\) edges, which shows that \(f_G\) has at least \(2\left\lfloor \frac{1}{12}n \right\rfloor\) subfunctions of order \(\frac{1}{2}n^2\).

**Proof that coset states need exponentially large QMDDs.** We now show that QMDDs which represent coset states are exponentially large in the worst case. We will use the following result by ŠDuriš et al. on binary decision diagrams (BDDs), which are QMDDs with codomain \(\{0,1\}\). This result concerns vector spaces, but of course, every vector space of \(\{0,1\}^n\) is, in particular, a coset.

**Theorem 10** (ŠDuriš et al. [81]). The characteristic function \(f_V : \{0,1\}^n \rightarrow \{0,1\}\) of a randomly chosen vector space \(V\in\{0,1\}^n\), defined as \(f_V(x) = 1\) if \(x\in V\) and 0 otherwise, needs a BDD of size \(2^{\Omega(n)}/(2n)\) with high probability.

Our result follows by noting that if \(f\) has codomain \(\{0,1\}\) as above, then the QMDD of the state \(|f\rangle = \sum_x f(x) |x\rangle\) has the same structure as the BDD of \(f\). Consequently, in particular the BDD and QMDD have the same number of nodes.

**Corollary 11.** For a random vector space \(V \subseteq \{0,1\}^n\), the coset state \(|V\rangle\) requires QMDDs of size \(2^{\Omega(n)}/(2n)\) with high probability.

**Proof.** We will show that the QMDD has the same number of nodes as a BDD. A BDD encodes a function \(f : \{0,1\}^n \rightarrow \{0,1\}\). In this case, the BDD encodes \(f_V\), the characteristic function of \(V\). A BDD is a graph which contains one node for each subfunction of \(f\). (In the literature, such a BDD is sometimes called a Full BDD, so that the term BDD is reserved for a variant where the nodes are in one-to-one correspondence with the subfunctions \(f\) which satisfy \(f_0 \neq f_1\)).

Similarly, a QMDD representing a state \(|\varphi\rangle = \sum_x f(x) |x\rangle\) can be said to represent the function \(f : \{0,1\}^n \rightarrow \mathbb{C}\), and contains one node for each subfunction of \(f\) modulo scalars. We will show that, two distinct subfunctions of \(f_V\) are never equal up to a scalar. To this end, let \(f_{V,a}, f_{V,b}\) be distinct subfunctions of \(f_V\) induced by partial assignments \(a, b \in \{0,1\}^k\). We will show that there is no \(\lambda \in \mathbb{C}\) such that \(f_{V,a} = \lambda f_{V,b}\). Since the two subfunctions are not pointwise equal, say that the two subfunctions differ in the point \(x \in \{0,1\}^{n-k}\), i.e., \(f_{V,a}(x) \neq f_{V,b}(x)\). Say without loss of generality that \(f_{V,a}(x) = 0\) and \(f_{V,b}(x) = 1\). Then, since \(\lambda \neq 0\), we have \(\lambda = \lambda f_{S,b}(x) \neq f_{V,a}(x) = 0\), so \(f_{V,a} \neq \lambda f_{S,b}\).
Because distinct subfunctions of $f_V$ are not equal up to a scalar, the QMDD of $|V\rangle$ contains a node for every unique subfunction of $f_V$. We conclude that, since by Th. 10 with high probability the BDD representing $f_V$ has exponentially many nodes, so does the QMDD representing $|V\rangle$.

C How to write graph states, coset states and stabilizer states as Tower-LIMDDs

In this appendix, we prove that the families of $\langle Z\rangle$, $\langle X\rangle$, and (Pauli)-Tower-LIMDDs correspond to graph states, coset states, and stabilizer states, respectively, in Th. 12, Th. 13 and Th. 1 below. Def. 5 for reduced Pauli-LIMDDs requires modification for $G = \langle Z\rangle$-LIMDDs because of the absence of $X$ as discussed below the definition. Note that the proofs do not rely on the specialized definition of reduced LIMDDs, but only on Def. 2 which allows parameterization of the LIM $G$. They only rely on the Tower LIMDD in Def. 3.

Before we give the proof, we remark that graph states present an interesting special case because the LIMDD’s edge labels contain meaningful information. Namely, the labels on the high edges of a graph state’s LIMDD are precisely the edges in the original graph. Specifically, suppose a $G$-LIMDD gives rise to a graph state $|\varphi_G\rangle$ represented by a LIMDD. Let $P = P_{k-1} \cdots P_1$ be the label on the high edge out of the LIMDD node at level $k$. Then $G$ contains an edge $(u, v)$ if and only if $P_j = Z$ (with the roles of $k$ and $j$ reversed if $k < j$). These edge labels come about in a straightforward manner during the construction of the graph state. Namely, the graph state $|\varphi_G\rangle$ is produced by starting from the state $|\rangle^{\otimes n}$, and applying controlled-$Z$ gates to qubit pairs $(u, v)$ for every edge $(u, v)$ in the graph. Applying such a controlled-$Z$ gate to qubit pair $(u, v)$ has the effect of setting $P_u$ to $Z$ in the high edge outgoing from the vertex at level $u$. In general, however, the labels on the high edges cannot be easily inferred from the stabilizer state.

A $G$-Tower-LIMDD representing an $n$-qubit state is a LIMDD which has $n$ nodes, not counting the leaf. It has $G$-LIMs on its high edges. Def. 3 gives an exact definition.

**Theorem 12** (Graph states are $\langle Z\rangle$-Tower-LIMDDs). Let $n \geq 1$. Denote by $G_n$ the set of $n$-qubit graph states and write $Z_n$ for the set of $n$-qubit quantum states which are represented by $\langle Z\rangle$-Tower-LIMDDs as defined in Def. 3, i.e., a tower with low-edge-labels $\lambda$ and high-edge labels $\lambda \otimes_j P_j$ with $P_j \in \{1, Z\}$ and $\lambda = 1$, except for the root edge where $\lambda \in \mathbb{C} \setminus \{0\}$. Then $G_n = Z_n$.

**Proof.** We establish $G_n \subseteq Z_n$ by providing a procedure to convert any graph state in $G_n$ to a $\langle Z\rangle$-Tower-LIMDD in $Z_n$. See Fig. 14 for an example of a 4-qubit graph state. We describe the procedure by induction on the number $n$ of qubits in the graph state.

**Base case:** $n = 1$. We note that there is only one single-qubit graph state by definition (see Eq. 11), which is $|\rangle := (|0\rangle + |1\rangle)/\sqrt{2}$ and can be represented as LIMDD by a single node (in addition to the leaf node): see Fig. 14(a).

**Induction case.** We consider an $(n+1)$-qubit graph state $|G\rangle$ corresponding to the graph $G$. We isolate the $(n+1)$-th qubit by decomposing the full state definition from Eq. 11:

$$
|G\rangle = \frac{1}{\sqrt{2}} \left( |0\rangle \otimes |G_{1..n}\rangle + |1\rangle \otimes \bigotimes_{(n+1),j \in E} Z_j \right) |G_{1..n}\rangle
$$

(15)

where $E$ is the edge set of $G$ and $G_{1..n}$ is the induced subgraph of $G$ on vertices $1$ to $n$. Thus, $|G_{1..n}\rangle$ is an $n$-qubit graph state on qubits $1$ to $n$. Since $|G_{1..n}\rangle$ is a graph state on $n$ qubits, by the induction hypothesis, we have a procedure to convert it to a $\langle Z\rangle$-Tower-LIMDD $\in Z_n$. Now we construct a $\langle Z\rangle$-Tower-LIMDD for $|G\rangle$ as follows. The root node has two outgoing edges, both
Figure 14: Construction of the \(\langle Z\rangle\)-Tower LIMDD for the 4-qubit cluster state, by iterating over the vertices in the graph, as described in the proof of Th. 12. (a) First, we consider the single-qubit graph state, which corresponds to the subgraph containing only vertex \(A\). (b) Then, we add vertex \(B\), which is connected to \(A\) by an edge. The resulting LIMDD is constructed from the LIMDD from (a) by adding a new root node. In the figure, the isomorphism is \(Z_B \otimes I\), since vertex \(C\) is connected to vertex \(B\) (yielding the \(Z\) operator) but not to \(A\) (yielding the identity operator \(I\)). (c) This process is repeated for a third vertex \(C\) until we reach the LIMDD of the full 4-qubit cluster state (d). For comparison, (d) also depicts a regular QMDD for the same graph state, which has width 4 instead of 1 for the LIMDD.

To prove \(Z_n \subseteq G_n\), we show how to construct the graph corresponding to a given \(\langle Z\rangle\)-Tower LIMDD. Briefly, we simply run the algorithm outlined above in reverse, constructing the graph one node at a time. Here we assume without loss of generality that the low edge of every node is labeled \(I\).

**Base case.** The LIMDD node above the Leaf node, representing the state \(|+\rangle\), always represents the singleton graph, containing one node.

**Induction case.** Suppose that the LIMDD node \(k+1\) levels above the Leaf has a low edge labeled \(I\), and a high edge labeled \(P_k \otimes \cdots \otimes P_1\), with \(P_j = Z^{a_j}\) for \(j = 1 \ldots k\). Here by \(Z^{a_j}\) we mean \(Z^{0} = I\) and \(Z^{1} = Z\). Then we add a node labeled \(k+1\) to the graph, and connect it to those nodes \(j\) with \(a_j = 1\), for \(j = 1 \ldots k\). The state represented by this node is of the form given in Eq. 15, so it represents a graph state.

A simple counting argument based on the above construction shows that \(|Z_n| = |G_n| = 2^{\binom{n}{2}}\), so the conversion is indeed a bijection. Namely, there are \(2^{\binom{n}{2}}\) graphs, since there are \(\binom{n}{2}\) edges to choose, and there are \(2^{\binom{n}{2}}\) \(\langle Z\rangle\)-Tower-LIMDDs, because the total number of single-qubit operators of the LIMs on the high edges is \(\binom{n}{2}\), each of which can be independently chosen to be either \(I\) or \(Z\).

We now prove that coset states are represented by \(\langle X\rangle\)-Tower-LIMDDs.

**Theorem 13** (coset states are \(\langle X\rangle\)-Tower-LIMDDs). Let \(n \geq 1\). Denote by \(\mathcal{V}_n\) the set of \(n\)-qubit coset states and write \(X_n\) for the set of \(n\)-qubit quantum states which are represented by \(\langle X\rangle\)-Tower-LIMDDs as per Def. 3, i.e., a tower with low edge labels \(I\) and high edge labels \(\lambda \otimes j P_j\) with \(P_j \in \{I, X\}\) and \(\lambda \in \{0, 1\}\), except for the root edge where \(\lambda \in \mathbb{C} \setminus \{0\}\). Then \(\mathcal{V}_n = X_n\).
Proof. We first prove \( V_n \subseteq X_n \) by providing a procedure for constructing a Tower-LIMDD for a coset state. We prove the statement for the case when \( C \) is a group rather than a coset; the result will then follow by noting that, by placing the label \( X^{a_n} \otimes \cdots \otimes X^{a_1} \) on the root edge, we obtain the coset state \( \langle C + a \rangle \). The procedure is recursive on the number of qubits.

**Base case:** \( n = 1 \). In this case, there are two coset states: \( |0\rangle \) and \( (|0\rangle + |1\rangle)/\sqrt{2} \), which are represented by a single node which has a low and high edge pointing to the leaf node with low/high edge labels \( 1 \) and \( 0 \), respectively.

**Induction case.** Now consider an \( (n+1) \)-qubit coset state \( |S\rangle \) for a group \( S \subseteq \{0,1\}^{n+1} \) for some \( n \geq 1 \) and assume we have a procedure to convert any \( n \)-qubit coset state into a Tower-LIMDD in \( X_n \). We consider two cases, depending on whether the first bit of each element of \( S \) is zero:

(a) The first bit of each element of \( S \) is 0. Thus, we can write \( S = \{0x \mid x \in S_0\} \) for some set \( S_0 \subseteq \{0,1\}^n \). Then \( 0a, 0b \in S \implies a, b \in S_0 \implies a \oplus b \in S_0 \) and thus \( S_0 \) is an \( n \)-bit string vector space. Thus by assumption, we have a procedure to convert it to a Tower-LIMDD in \( X_n \). Convert it into a Tower-LIMDD in \( X_{n+1} \) for \( |S\rangle \) by adding a fresh node on top with low edge label \( \mathbb{I}^{\otimes n} \) and high edge label 0, both pointing to the root node \( S \).

(b) There is some length-\( n \) bit string \( u \) such that \( 1u \in S \). Write \( S \) as the union of the sets \( \{0x \mid x \in S_0\} \) and \( \{1x \mid x \in S_1\} \) for sets \( S_0, S_1 \subseteq \{0,1\}^n \). Since \( S \) is closed under element-wise XOR, we have \( 1u \oplus 1x = 0(u \oplus x) \in S \) for each \( x \in S_1 \) and therefore \( u \oplus x \in S_0 \) for each \( x \in S_1 \). This implies that \( S_1 = \{u \oplus x \mid x \in S_0\} \) and thus \( S \) is the union of \( \{0x \mid x \in S_0\} \) and \( \{1u \oplus 0x \mid x \in S_0\} \). By similar reasoning as in case (a), we can show that \( S_0 \) is a vector space on length-\( n \) bit strings.

We build a Tower-LIMDD for \( |S\rangle \) as follows. By the induction hypothesis, there is a Tower-LIMDD with root node \( v \) which represents \( |v\rangle = |S_0\rangle \). We construct a new node whose two outgoing edges both go to this node \( v \). Its low edge has label \( \mathbb{I}^{\otimes n} \) and its high edge has label \( P = P_n \otimes \cdots \otimes P_1 \) where \( P_j = X \) if \( u_j = 1 \) and \( P_j = \mathbb{I} \) if \( u_j = 0 \).

We now show \( V_n \subseteq X_n \), also by induction.

**Base case:** \( n = 1 \). There are only two Tower-LIMDDs on 1 qubit satisfying the description above, namely

(1) A node whose two edges point to the leaf. Its low edge has label 1, and its high edge has label 0. This node represents the coset state \( |0\rangle \), corresponding to the vector space \( V = \{0\} \subseteq \{0,1\}^2 \).

(2) A node whose two edges point to the leaf. Its low edge has label 1 and its high edge also has label 1. This node represents the coset state \( |0 \rangle + |1 \rangle \), corresponding to the vector space \( V = \{0,1\} \).

**Induction case.** Let \( v \) be the root node of an \( n+1 \)-qubit Tower \( \langle X \rangle \)-LIMDD as described above. We distinguish two cases, depending on whether \( v \)’s high edge has label 0 or not.

(a) The high edge has label 0. Then \( |v\rangle = |0\rangle \langle v_0| \) for a node \( v_0 \), which represents a coset state \( |v_0\rangle \) corresponding to a coset \( V_0 \subseteq \{0,1\}^n \), by the induction hypothesis. Then \( v \) corresponds to the coset \( \{0x \mid x \in V_0\} \).

(b) the high edge has label \( P = P_n \otimes \cdots \otimes P_1 \) with \( P_j \in \{\mathbb{I}, X\} \). Then \( |v\rangle = |0\rangle \langle v_0| + |1\rangle \otimes P |v_0\rangle \). By the observations above, this is a coset state, corresponding to the vector space \( V = \{0x \mid x \in V_0\} \cup \{1(u \oplus x) \mid x \in V_0\} \) where \( u \in \{0,1\}^n \) is a string whose bits are \( u_j = 1 \) if \( P_j = X \) and \( u_j = 0 \) if \( P_j = \mathbb{I} \), and \( V_0 \) is the vector space corresponding to the coset state \( |v_0\rangle \). 

Lastly, we prove the stabilizer-state case, showing that they are exactly equivalent to the \( (\text{PAULI}) \)-Tower-LIMDD, as defined in Def. 3. For this, we first need Lemma 3 and Lemma 4, which state that,
if one applies a Clifford gate to a (Pauli)-Tower-LIMDD, the resulting state is another (Pauli)-Tower-LIMDD. First, Lemma 3 treats the special case of applying a gate to the top qubit; then Lemma 4 treats the general case of applying a gate to an arbitrary qubit.

**Lemma 3.** Let $|φ⟩$ be an $n$-qubit stabilizer state which is represented by a (Pauli)-Tower-LIMDD as defined in Def. 3. Let $U$ be either a Hadamard gate or $S$ gate on the top qubit ($n$-th qubit), or a downward CNOT with the top qubit as control. Then $U |φ⟩$ is still represented by a (Pauli)-Tower-LIMDD.

*Proof.* The proof is on the number $n$ of qubits.

**Base case:** $n = 1$. For $n = 1$, there are six single-qubit stabilizer states $|0⟩, |1⟩$ and $(|0⟩ + |1⟩)/\sqrt{2}$ for $α \in \{±1, ±i\}$. There are precisely represented by Pauli-Tower-LIMDDs with high edge label factor $∈ \{0, ±1, ±i\}$ as follows:

- for $|0⟩$: $\text{□} \xrightarrow{X} \text{□}$
- for $|1⟩$: $A \cdot \text{□} \xrightarrow{X} \text{□}$ where $A ∝ X$ or $A ∝ Y$
- for $(|0⟩ + |1⟩)/\sqrt{2}$: $\text{□} \xrightarrow{X} \text{□}$

Since the $H$ and $S$ gate permute these six stabilizer states, $U |φ⟩$ is represented by a (Pauli)-Tower-LIMDD if $|φ⟩$ is.

**Induction case.** For $n > 1$, we first consider $U = S$ and $U = CNOT$. Let $R$ be the label of the root edge. If $U = S$, then the high edge of the top node is multiplied with $i$, while a downward CNOT (target qubit with index $k$) updates the high edge label $A → X_k A$. Next, the root edge label is updated to $URU^†$, which is still a Pauli string, since $U$ is a Clifford gate. Since the high labels of the top qubit in the resulting diagram is still a Pauli string, and the high edge’s weights are still $∈ \{0, ±1, ±i\}$, we conclude that both these gates yield a (Pauli)-Tower-LIMDD. Finally, for the Hadamard, we decompose $|φ⟩ = |0⟩ ⊕ |ψ⟩ + α |1⟩ ⊕ P|ψ⟩$ for some $(n - 1)$-qubit stabilizer state $|ψ⟩$, $α ∈ \{0, ±1, ±i\}$ and $P$ is an $(n - 1)$-qubit Pauli string. Now we note that $H |φ⟩ ∝ |0⟩ ⊕ |ψ⟩ + |1⟩ ⊕ |ψ⟩$ where $|ψ⟩ := (1 + (-1)^{x} α P) |ψ⟩$ with $x ∈ \{0, 1\}$. Now we consider two cases, depending on whether $P$ commutes with all stabilizers of $|ψ⟩$:

(a) There exist a stabilizer $g$ of $|ψ⟩$ which anticommutes with $P$. We note two things. First, $⟨ψ|P|ψ⟩ = ⟨ψ|P|ψ⟩ = ⟨ψ|g \cdot (−P)|ψ⟩ = −⟨ψ|P|ψ⟩$, hence $⟨ψ|P|ψ⟩ = 0$. It follows from Lemma 15 of [82] that $|ψ⟩$ is a stabilizer state, so by the induction hypothesis it can be written as a (Pauli)-Tower-LIMDD. Let $v$ be the root node of this LIMDD. Next, we note that $g |ψ⟩ = g (I + α P) |ψ⟩ = (I − α P) g |ψ⟩ = |ψ⟩$. Hence, $\text{□} \xrightarrow{X} \text{□}$ is the root node of a (Pauli)-Tower-LIMDD for $H |φ⟩$.

(b) All stabilizers of $|ψ⟩$ commute with $P$. Then $(-1)^{y} P$ is a stabilizer of $|ψ⟩$ for either $y = 0$ or $y = 1$. Hence, $|ψ⟩ = (1 + (-1)^{y} α P) |ψ⟩ = (1 + (-1)^{y} α) |ψ⟩$. Therefore, $|φ⟩ = |α⟩ ⊕ |ψ⟩$ where $|α⟩ := (1 + (-1)^{y} α) |0⟩ + (1 + (-1)^{y+1} α) |1⟩$. It is not hard to see that $|α⟩$ is a stabilizer for all choices of $α ∈ \{0, ±1, ±i\}$. By the induction hypothesis, both $|α⟩$ and $|ψ⟩$ can be represented as (Pauli)-Tower-LIMDDs. We construct a (Pauli)-Tower-LIMDD for $H |φ⟩$ by replacing the leaf of the LIMDD of $|α⟩$ by the root node of the LIMDD of $|ψ⟩$, and propagating the root edge label of $|ψ⟩$ upwards. Specifically, if the root edge of $|α⟩$ is $A$ with $v = \text{□} \xrightarrow{X} \text{□}$, and if the root edge of $|ψ⟩$ is $B$, then a (Pauli)-Tower-LIMDD for $H |φ⟩$ has root node $\text{□} \xrightarrow{X} \text{□}$ and has root edge label $A ⊗ B$. 

**Lemma 4.** Let $|φ⟩$ be an $n$-qubit state state represented by a (Pauli)-Tower-LIMDD, as defined in Def. 3. Let $U$ be either a Hadamard gate, an $S$ gate or a CNOT gate. Then $U |φ⟩$ is a state which is also represented by a (Pauli)-Tower-LIMDD.
Proof. The proof is by induction on \( n \). The case \( n = 1 \) is covered by Lemma 3. Suppose that the induction hypothesis holds, and let \( |\varphi\rangle \) be an \( n + 1 \)-qubit state represented by a \((\text{Pauli})\)-Tower-LIMDD. First, we note that a CNOT gate \( CX_1^t \) can be written as \( CX_1^t = (H \otimes H)CX_1^s(H \otimes H) \), so without loss of generality we may assume that \( c > t \). We treat two cases, depending on whether \( U \) affects the top qubit or not.

(a) \( U \) affects the top qubit. Then \( U |\varphi\rangle \) is represented by a \((\text{Pauli})\)-Tower-LIMDD, according to Lemma 3.

(b) \( U \) does not affect the top qubit. Suppose \( |\varphi\rangle = |0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes P |\varphi_0\rangle \) (with \( P \) a Pauli string and \( \alpha \in \{0, \pm 1, \pm i\} \)). Then \( U |\varphi\rangle = |0\rangle \otimes U |\varphi_0\rangle + |1\rangle \otimes (U\alpha U^\dagger) U |\varphi_0\rangle \). Since \( U \) is either a Hadamard, \( S \) gate or CNOT, and \( |\varphi_0\rangle \) is an \( n \)-qubit state, the induction hypothesis states that the state \( U |\varphi_0\rangle \) is represented by a \((\text{Pauli})\)-Tower-LIMDD. Let \( \lambda P \) be the root edge of this \((\text{Pauli})\)-Tower-LIMDD, representing \( U |\varphi_0\rangle \). Then \( U |\varphi\rangle \) is represented by the root edge \( 1 \otimes \lambda P \), where \( w \) is the node \( \bigodot \lambda P \bigodot 1 \otimes C^{-1} UPU^\dagger A \). The label \( \alpha A^{-1} UPU^\dagger A \) is a Pauli LIM, and may therefore be used as the label on the high edge of \( w \).

Finally, we show that stabilizer states are precisely the \((\text{Pauli})\)-Tower-LIMDDs.

**Theorem 1.** Let \( n > 0 \). Each \( n \)-qubit stabilizer state is represented up to normalization by a \((\text{Pauli})\)-Tower-LIMDDs of Def. 3, e.g., where the scalars \( \lambda \) of the PauliLIMs \( A \) on high edges are restricted as \( \lambda \in \{0, \pm 1, \pm i\} \). Conversely, every such LIMDD represents a stabilizer state.

**Proof.** We first prove that each stabilizer state is represented by a \((\text{Pauli})\)-Tower-LIMDD. We recall that each stabilizer state can be obtained as the output state of a Clifford circuit on input state \( |0\rangle \otimes |0\rangle \). Each Clifford circuit can be decomposed into solely the gates \( H, S \) and CNOT. The state \( |0\rangle \otimes |0\rangle \) is represented by a \((\text{Pauli})\)-Tower-LIMDD. According to Lemma 4, applying an \( H, S \) or CNOT gate to a \((\text{Pauli})\)-Tower-LIMDD results a state represented by another \((\text{Pauli})\)-Tower-LIMDD. One can therefore apply the gates of a Clifford circuit to the initial state \( |0\rangle \), and obtain a \((\text{Pauli})\)-Tower-LIMDD for every intermediate state, including the output state. Therefore, every stabilizer state is represented by a \((\text{Pauli})\)-Tower-LIMDD.

For the converse direction, the proof is by induction on \( n \). We only need to note that a state represented by a \((\text{Pauli})\)-Tower-LIMDD can be written as \( |\varphi\rangle = |0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes \alpha P |\varphi_0\rangle = C(P)(|0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes \alpha P |\varphi_0\rangle) \) where \( C(P) := |0\rangle \otimes \mathbb{I} + |1\rangle \otimes P \) is the controlled-\( (P) \) gate. Using the relations \( Z = HXH, Y = SXS^t \) and \( S = Z^2 \), we can decompose \( C(P) \) as CNOT, \( H \) and \( S \), hence \( C(P) \) is a Clifford gate. Since both \( |0\rangle + |1\rangle \) and \( |\varphi_0\rangle \) can be written as \((\text{Pauli})\)-Tower-LIMDDs, they are stabilizer states by the induction hypothesis. Therefore, the state \( |\psi\rangle = (|0\rangle + |1\rangle) \otimes |\varphi_0\rangle \) is also a stabilizer state. Thus, the state \( |\varphi\rangle = C(P)|\psi\rangle \) is obtained by applying the Clifford gate \( C(P) \) to the stabilizer state \( |\varphi\rangle \). Therefore, \( |\varphi\rangle \) is a stabilizer state.

**D** Efficient algorithms for choosing a canonical high label

Here, we present an efficient algorithm which, on input Pauli-LIMDD node \( \bigodot x^t \bigodot \lambda P \bigodot 1 \), returns a canonical choice for the high label \( B_{\text{high}} \) (algorithm \text{GETLABELS}, in Alg. 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 \( |v\rangle \simeq_{\text{Pauli}} |w\rangle \).

We first characterize all eligible labels \( B_{\text{high}} \) 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 Sec. 2 for the definition of stabilizer subgroup). Then, we provide the algorithm \text{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.
Th. 14 shows that this way to choose the high label indeed captures all eligible high labels, i.e., a node \( \bigotimes_1^w \lambda P \) is isomorphic to \( |w\rangle \) if and only if \( B_{\text{high}} \) is of this form.

**Theorem 14** (Eligible high-edge labels). Let \( \bigotimes_1^w \lambda P \) 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 \( v = \bigotimes_1^w \lambda P \), it holds that \( |w\rangle \simeq |v\rangle \) if and only if

\[
B_{\text{high}} = (-1)^x \cdot \lambda^{(-1)^x} g_0 P g_1
\]

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

\[
B_{\text{root}} = (X \otimes \lambda P)^x \cdot (Z^* \otimes (g_0)^{-1}).
\]

**Proof.** It is straightforward to verify that the isomorphism \( B_{\text{root}} \) in eq. (18) 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.,

\[
C ((0) \otimes |v_0\rangle + \lambda |1 \rangle \otimes |v_1\rangle) = (0) \otimes |v_0\rangle + |1 \rangle \otimes B_{\text{high}} |v_1\rangle.
\]

We show that if \( B_{\text{high}} \) satisfies eq. (19), then it has a decomposition as in eq. (17). We write \( C = C_{\text{top}} \otimes C_{\text{rest}} \) where \( C_{\text{top}} \) is a single-qubit Pauli operator and \( C_{\text{rest}} \) is an \((n - 1)\)-qubit Pauli LIM (or a complex number \( \neq 0 \) if \( n = 1 \)). We treat the two cases \( C_{\text{top}} \in \{I, Z\} \) and \( C_{\text{top}} \in \{X, Y\} \) separately:

(a) **Case** \( C_{\text{top}} \in \{I, Z\} \). Then \( C_{\text{top}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & (-1)^y \end{pmatrix} \) for \( y \in \{0, 1\} \). In this case, Eq. (19) implies \( C_{\text{top}} |0\rangle = |0\rangle \), so \( C_{\text{rest}} |v_0\rangle = |v_0\rangle \), in other words \( C_{\text{rest}} \in \text{Stab}(|v_0\rangle) \). Moreover, Eq. (19) implies \((-1)^y\lambda^{(-1)^{y-1}}C_{\text{rest}} B_{\text{high}} |v_1\rangle = B_{\text{high}} |v_1\rangle \), or, equivalently, \((-1)^{y-1} \lambda^{-1} P^{-1} C_{\text{rest}} B_{\text{high}} \in \text{Stab}(|v_1\rangle) \). Hence, by choosing \( s = y \) and \( x = 0 \), we compute

\[
(-1)^y \lambda^{(-1)^y} C_{\text{rest}} \bigoplus_{\text{Stab}(v_0)} P (\lambda)^{(-1)^y \lambda^{-1} P^{-1} C_{\text{rest}} B_{\text{high}}) = (-1)^y \lambda^{(-1)^y} (-1)^y \lambda B_{\text{high}} = B_{\text{high}}.
\]

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

Figure 15: Illustration of finding a canonical high label for a semi-reduced node \( w \), yielding a reduced node \( v' \). 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 \( \lambda^{(-1)^y} \) for some \( x \in \{0, 1\} \). There are two cases: if \( v_0 \neq v_1 \) or \( x = 0 \), then the factor is \( \lambda \) and the root edge should be adjusted with an \( I \) or \( Z \) on the root qubit. The other case, \( x = 1 \), leads to an additional multiplication with an \( X \) on the root qubit.
(b) Case $C_{\text{top}} \in \{X,Y\}$. Write $C_{\text{top}} = \left[\begin{array}{c} 0 & z^{-1} \\ z & 0 \end{array}\right]$ where $z \in \{1,i\}$. Now, eq. (19) implies

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

(20)

From Eq. 20, we first note that $|v_0\rangle$ and $|v_1\rangle$ are isomorphic, so by Corollary 9, and because the diagram has merged these two nodes, we have $v_0 = v_1$. Consequently, we find from Eq. 20 that $z^{-1}C_{\text{rest}}B_{\text{high}} \in \text{Stab}(v_0)$ and $z^{-1}\lambda C_{\text{rest}}P \in \text{Stab}(v_1)$. Now choose $x = 1$ and choose $s$ such that $(-1)^s \cdot z^{-2}C_{\text{rest}}B_{\text{high}}C_{\text{rest}} = B_{\text{high}}$ (recall that Pauli LIMs either commute or anticommute, so $B_{\text{high}}C_{\text{rest}} = \pm C_{\text{rest}}B_{\text{high}}$). This yields:

$$(-1)^s \lambda^{-1} \cdot z^{-1}C_{\text{rest}}B_{\text{high}} \cdot \lambda^{-1} \cdot P \cdot C_{\text{rest}} = \lambda^{-1} \cdot (1)^s \cdot z^{-2} \cdot (C_{\text{rest}}B_{\text{high}}C_{\text{rest}}) = B_{\text{high}}$$

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

\[ \square \]

Corollary 15. As a corollary of Th. 14, we find that taking, as in Fig. 15,

$$\text{HighLabel}(\overline{\text{X}}^1 \cdot \overline{\text{O}} \cdot \overline{\text{L}} \cdot \overline{\text{P}} \cdot \overline{\text{O}}) = \min_{i,s,x \in \{0,1\},g_0 \in \text{Stab}(v_1)} \left\{ \left\{ (-1)^s \cdot \lambda^{-1} \cdot g_0 \cdot P \cdot g_1 \mid x \neq 1 \text{ if } v_0 \neq v_1 \right\} \right\}$$

yields a proper implementation of $\text{HighLabel}$ as required by Def. 5, because it considers all possible $B_{\text{high}}$ such that $|v \rangle \sim_{\text{PAULI}} |0\rangle |v_0\rangle + |1\rangle \otimes B_{\text{high}}|v_1\rangle$.

A naive implementation for $\text{GetLabels}$ would follow the possible decompositions of eligible LIMs (see Eq. 17) and attempt to make this LIM smaller by greedy multiplication, first with stabilizers of $g_0 \in \text{Stab}(v_0)$, and then with stabilizers $g_1 \in \text{Stab}(v_1)$. To see why this does not work, consider the following example: the high edge label is $Z$ and the stabilizer subgroups are $\text{Stab}(v_0) = \langle X \rangle$ and $\text{Stab}(v_1) = \langle Y \rangle$. Then the naive algorithm would terminate and return $Z$ because $X \cdot Z \cdot Y = -I$ is smaller than $Z$.

Algorithm 12. Algorithm for finding LIMs $B_{\text{high}}$ and $B_{\text{root}}$ required by $\text{MAKEEdge}$. Its parameters represent a semi-reduced node $\overline{\text{X}}^1 \cdot \overline{\text{O}} \cdot \lambda \cdot \overline{\text{P}} \cdot \overline{\text{O}}$ and it returns LIMs $B_{\text{high}}, B_{\text{root}}$ such that $|v\rangle = B_{\text{root}}|w\rangle$ with $\overline{\text{X}}^1 \cdot \overline{\text{O}} \cdot B_{\text{high}} \cdot \overline{\text{O}}$. The LIM $B_{\text{high}}$ is chosen canonically as the lexicographically smallest from the set characterized in Th. 14. It runs in $O(n^3)$-time (with $n$ the number of qubits), provided $\text{GetStabilizerGenSet}$ has been computed for children $v_0, v_1$ (an amortized cost).

1: \textbf{procedure} $\text{GetLabels}$(PAULILIM $\lambda \cdot \text{AP}$, NODE $v_0, v_1$ with $\lambda \neq 0$ and $v_0, v_1$ reduced) 
\hspace{1cm} \textbf{Output}: canonical high label $B_{\text{high}}$ and root label $B_{\text{root}}$
2: \hspace{1cm} $G_0, G_1 := \text{GetStabilizerGenSet}(v_0), \text{GetStabilizerGenSet}(v_1)$
3: \hspace{1cm} $(g_0, g_1) := \text{argLexMin}(G_0, G_1, \lambda P)$
4: \hspace{1cm} \textbf{if} $v_0 = v_1$ \textbf{then}
5: \hspace{1.5cm} $(x, s) := \text{arg min}_{(x,s) \in \{0,1\}^2} \{-(-1)^s \cdot \lambda g_0 P g_1\}$
6: \hspace{1cm} \textbf{else}
7: \hspace{2cm} $x := 0$
8: \hspace{2cm} $s := \text{arg min}_{s \in \{0,1\}} \{-(-1)^s \lambda g_0 P g_1\}$
9: \hspace{1cm} $B_{\text{high}} := (-1)^s \cdot \lambda (-1)^s \cdot g_0 \cdot P \cdot g_1$
10: \hspace{1cm} $B_{\text{root}} := (X \otimes \text{AP})^x \cdot (Z^s \otimes (g_0)^{-1})$
11: \hspace{1cm} \textbf{return} $(B_{\text{high}}, B_{\text{root}})$

To overcome this, we consider the group closure of both $\text{Stab}(v_0)$ and $\text{Stab}(v_1)$. See Alg. 12 for the $O(n^3)$-algorithm for $\text{GetLabels}$, which proceeds in two steps. In the first step (Line 3), we
use the subroutine \textsc{ArgLexMin} for finding the minimal Pauli LIM A such that $A = \lambda P \cdot g_0 \cdot g_1$ for $g_0 \in \text{Stab}(v_0), g_1 \in \text{Stab}(v_1)$. We will explain and prove correctness of this subroutine below in Sec. D.2. In the second step (Line 4-8), we follow Corollary 15 by also minimizing over $x$ and $s$. Finally, the algorithm returns $B_{\text{high}}$, the minimum of all eligible edge labels according to Corollary 15, together with a root edge label $B_{\text{root}}$ 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 \textsc{ArgLexMin}. Since all other lines in Alg. 12 can be performed in linear time, its overall runtime is $O(n^3)$.

D.1 Constructing the stabilizer subgroup of a LIMDD node

In this section, we give a recursive subroutine \textsc{GetStabilizerGenSet} to construct the stabilizer subgroup $\text{Stab}(v) = \{A \in \text{PauliLIM}_n \mid \text{A}(v) = |v\rangle\}$ of an $n$-qubit LIMDD node $v$ (see Sec. 2). This subroutine is used by the algorithm \textsc{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, \textsc{GetStabilizerGenSet}’s runtime is $O(n^3)$. \textsc{GetStabilizerGenSet} returns a generating set for the group $\text{Stab}(v)$. 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 $O(|G| \cdot n^2)$ (see App. A), 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 $v = \bigotimes_i \{0\} \bigotimes B_{\text{high}} \bigotimes \{0\}$. If $B_{\text{high}} = 0$, then it is straightforward to see that $\lambda P_n \otimes P' \otimes |v\rangle = |v\rangle$ implies $P_n \in \{I, Z\}$, and further that $\text{Stab}(v) = \{\{P_n \otimes g \mid g \in G_0, P_n \in \{I, Z\}\}\}$, where $G_0$ is a stabilizer generator set for $v_0$.

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

\[
\lambda P_n \otimes P'(0) \otimes |v_0\rangle + (1) \otimes B_{\text{high}} |v_1\rangle = |0\rangle \otimes |v_0\rangle + (1) \otimes B_{\text{high}} |v_1\rangle ,
\]

which implies:

\[
\lambda P' |v_0\rangle = |v_0\rangle \quad \text{and} \quad z \lambda P' B_{\text{high}} |v_1\rangle = B_{\text{high}} |v_1\rangle \quad \text{if} \quad P_n = \left[ \begin{array}{c} 1 & 0 \\ 0 & 1 \end{array} \right] \quad \text{with} \quad z \in \{1, -1\} \quad (21)
\]

\[
y^* \lambda P' B_{\text{high}} |v_1\rangle = |v_0\rangle \quad \text{and} \quad \lambda P' |v_0\rangle = y^* B_{\text{high}} |v_1\rangle \quad \text{if} \quad P_n = \left[ \begin{array}{c} 0 & 1 \\ 1 & 0 \end{array} \right] , \quad \text{with} \quad y \in \{1, i\} \quad (22)
\]

The stabilizers can therefore be computed according to Eq. 21 and 22 as follows:

\[
\text{Stab}(v) = \bigcup_{z \in \{1, -1\}} \left\{ \left[ \begin{array}{c} 1 & 0 \\ 0 & 1 \end{array} \right] \otimes \text{Stab}(v_0) \cap z \cdot \text{Stab}(B_{\text{high}} |v_1\rangle) \right\} \bigcup \left\{ \left[ \begin{array}{c} 0 & 1 \\ 1 & 0 \end{array} \right] \otimes \text{Iso}(y^* B_{\text{high}} |v_1\rangle, |v_0\rangle) \cap \text{Iso}(|y^* B_{\text{high}} |v_1\rangle, |v_0\rangle) \right\} \quad (23)
\]

where $\text{Iso}(v, w)$ denotes the set of Pauli isomorphisms $A$ which map $|v\rangle$ to $|w\rangle$ and we have denoted $\pi \cdot G := \{\pi \cdot g \mid g \in G\}$ for a set $G$ and a single operator $\pi$. Lemma 5 shows that such an isomorphism set can be expressed in terms of the stabilizer group of $|v\rangle$.

\textbf{Lemma 5.} Let $|\varphi\rangle$ and $|\psi\rangle$ be quantum states on the same number of qubits. Let $\pi$ be a Pauli isomorphism mapping $|\varphi\rangle$ to $|\psi\rangle$. Then the set of Pauli isomorphisms mapping $|\varphi\rangle$ to $|\psi\rangle$ is $\text{Iso}(|\varphi\rangle, |\psi\rangle) = \pi \cdot \text{Stab}(|\varphi\rangle)$. That is, the set of isomorphisms $|\varphi\rangle \rightarrow |\psi\rangle$ is a coset of the stabilizer subgroup of $|\varphi\rangle$.

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

\hfill \square
With Lemma 5 we can rewrite eq. (23) as

\[
\text{Stab}(|v\rangle) = \mathbb{I} \otimes (\text{Stab}(|v_0\rangle) \cap \text{Stab}(B_{\text{high}} |v_1\rangle)) \\
\cup Z \otimes (\mathbb{I} \cdot \text{Stab}(|v_0\rangle) \cap -\mathbb{I} \cdot \text{Stab}(B_{\text{high}} |v_1\rangle)) \\
\cup \bigcup_{y \in \{1, i\}} \left[ \begin{array}{cc} 0 & y \\ y^* & 0 \end{array} \right] \otimes \left( \pi \cdot \text{Stab}(y^* B_{\text{high}} \cdot |v_1\rangle) \cap \pi^{-1} \cdot \text{Stab}(|v_0\rangle) \right)
\]

(24)

where \(\pi\) denotes a single isomorphism \(y^* B_{\text{high}} |v_1\rangle \rightarrow |v_0\rangle\).

Given generating sets for \(\text{Stab}(v_0)\) and \(\text{Stab}(v_1)\), evaluating eq. (24) requires us to:

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

- **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 Alg. 16.

- **Find the intersection of two stabilizer subgroups, represented as generating sets \(G_0\) and \(G_1\) (IntersectStabilizerGroups, Alg. 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 \(-1\). 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" 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 \(\lambda P\) 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 \(P \in G_0\) from a Pauli stabilizer group \(G_0\) 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 \(P \in G_0 \cap G_1\), i.e., in the group modulo phase, there exists a unique \(\lambda\) such that \(\lambda \cdot P \in \langle G_0 \rangle\); and a unique \(\omega\) 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 \(S = \{P_1, \ldots, P_t\}\) is a generating set for the group \((S) = \langle G_0 \rangle \cap \langle G_1 \rangle\) 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 \(\lambda \cdot P \in G_0 \cap G_1\) for some \(\lambda \in \mathbb{C}\) and each \(P \in S_1\) satisfies \(-\lambda \cdot P \in G_0\). 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 \(\{e_1 \cdot e_2 \cdot \ldots \cdot e_m\}\), found by 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 \(O(n^3)\) 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 \(F_2\), which is constructed in time \(O(n^4)\) on Line 3 using the Zassenhaus algorithm. On line Line 7, checking whether \(\lambda P \in G_0\) for given \(\lambda, P\) can be done in \(O(n^2)\) time; this happens at most \(O(|S|) = O(n)\) times, so in total this operation takes up \(O(n^3)\) time. Lastly, the loop in Line 14-16 runs in \(O(n^2)\) time, as there are at most \(|S| - 1 = O(n)\) multiplications of Pauli strings, each of which takes \(O(m)\) 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 \(G_1\).

---

1To 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 App. A.
The coset Lemma 6. \[ \Rightarrow \]
and finish the evaluation of eq. (24)
\[ I \]
The four algorithms above allow us to evaluate each of the four individual terms in eq. (24). To evaluate eq. (24), one would expect that it is also necessary that we find the union of isomorphism sets. However, we note that if \( \pi G \) is an isomorphism set, with \( \pi \) an isomorphism and \( G \) a stabilizer subgroup, then \( P_n \otimes (\pi G) = (P_n \otimes \pi)(I \otimes g) \) for all \( g \in G \). Therefore, we will evaluate eq. (24), 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. \( I \otimes (\text{Stab}([v_0]) \cap \text{Stab}([v_1])) \), 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 Alg. 13 and prove its efficiency below.

Lemma 7 (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 Alg. 13). Then Alg. 13 (function GetStabilizerGenSet), applied to \( v \), runs in time \( O(n^3) \).

Proof. If \( n = 1 \) then Alg. 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 Alg. 14 invokes \( O(n^3) \)-runtime external algorithms:

- the Zassenhaus algorithm [80] to calculate a basis for the intersection of two subspaces of a vector space,
- the RREF algorithms mentioned in App. A, and
• Algorithm 2 from [82] 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 $-I$. Our algorithms use this more general use case.

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

\[ \]

**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 App. A), which is accomplished in line 27. In particular, the set always returns at most $n$ elements for $n$-qubit states.

1: \begin{algorithmic}[1]
   \Procedure{GetStabilizerGenSet}{Edge $e = \begin{tikzpicture}[baseline]
      \node (e0) at (0,0) {$e_0$};
      \node (b) at (1,0) {$B$};
      \node (v0) at (2,0) {$v_0$};
      \node (v1) at (3,0) {$v_1$};
      \node (bhigh) at (4,0) {$B_{\text{high}}$};
      \node (e1) at (5,0) {$e_1$};
      \draw[-latex] (e0) to[bend right=90] (b);
      \draw[-latex] (b) to[bend right=90] (v0);
      \draw[-latex] (v0) to[bend right=90] (v1);
      \draw[-latex] (v1) to[bend right=90] (bhigh);
      \end{tikzpicture}$ with $v_0, v_1$ reduced}
   \State if $n = 1$ then
   \If there exists $P \in \{X, Y, Z\}$ such that $P|v⟩ = |v⟩$ then return $P$ \Else return None \EndIf
   \Else if $π ∈ \text{STABCACHE}[v]$ then return $\text{STABCACHE}[v]$ \EndIf
   \State $G_0 := \text{GetStabilizerGenSet}(v_0)$
   \State if $B_{\text{high}} = 0$ then
   \State $G := ∅$ \Else
   \State $G_1 := \{B_{\text{high}} \cdot g \cdot B_{\text{high}}^\dagger \mid g ∈ \text{GetStabilizerGenSet}(v_1)\}$
   \State $(π, B) := \text{IntersectIsomorphismSets}((1^n, G_0), (1^n, G_1))$
   \State $G := G \cup \{1^n \otimes g \mid g ∈ B\}$ ↪ Add all stabilizers of the form $1^n \otimes \ldots$
   \EndIf
   \State $π_0, π_1 := \text{GetIsomorphism}(e_1, -1 \cdot e_1)$
   \State $(π, B) := \text{IntersectIsomorphismSets}((π_0, G_0), (π_1, G_1))$
   \State if $π \neq \text{None}$ then $G := G ∪ \{Z \otimes π\}$ ↪ Add stabilizer of form $Z \otimes \ldots$
   \EndIf
   \State $π_0, π_1 := \text{GetIsomorphism}(e_0, e_1), \text{GetIsomorphism}(e_1, e_0)$
   \State $(π, B) := \text{IntersectIsomorphismSets}((π_0, G_0), (π_1, G_1))$
   \State if $π \neq \text{None}$ then $G := G ∪ \{X \otimes π\}$ ↪ Add stabilizer of form $X \otimes \ldots$
   \EndIf
   \State $π_0, π_1 := \text{GetIsomorphism}(e_0, -i \cdot e_1), \text{GetIsomorphism}(-i \cdot e_1, e_0)$
   \State $(π, B) := \text{IntersectIsomorphismSets}((π_0, G_0), (π_1, G_1))$
   \State if $π \neq \text{None}$ then $G := G ∪ \{Y \otimes π\}$ ↪ Add stabilizer of form $Y \otimes \ldots$
   \EndIf
   \State $G := \text{RREF}(G)$ ↪ Bring $G$ to reduced row echelon form, potentially pruning some rows
   \State $\text{STABCACHE}[v] := G$
   \State return $G$
\EndProcedure
\end{algorithmic}

\[ \]

D.2 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 $\min_{(g_0, g_1) ∈ (G_0, G_1)} A : g_0 \cdot g_1$, and also find the $g_0, g_1$ which minimize the expression. We give an algorithm for finding both the
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 5).

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

1: procedure INTERSECTISOMORPHISMSETS(stabilizer subgroup generating sets $G_0, G_1$, Pauli-LIMs $\pi_0, \pi_1$)

2: $\pi := LexMin(G_0, G_1, \pi_1^{-1} \pi_0)$

3: if $\pi = I$ then

4: $(g_0, g_1) = \text{ArgLexMin}(G_0, G_1, \pi_1^{-1} \pi_0)$

5: $\pi := \pi_0 \cdot g_0$

6: return $\langle \pi, G \rangle$

else

7: $\pi := \pi_0$

8: $G := \text{IntersectStabilizerGroups}(G_0, G_1)$

9: 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 $\langle G_0 \rangle \cap \langle G_1 \rangle$

1: procedure INTERSECTSTABILIZERGROUPS(stabilizer subgroup generating sets $G_0, G_1$)

2: Use the Zassenhaus algorithm to compute the intersection modulo phase

3: $S := \text{IntersectGroupsModuloPhase}(G_0, G_1)$

4: $S_0, S_1 := \emptyset$

5: for $P \in S$ do

6: By abuse of language, we treat $P$ as a Pauli string with phase $+1$

7: Find $\lambda \in \{\pm 1, \pm i\}$ such that $\lambda \cdot P \in \langle G_0 \rangle$

8: if $\lambda \cdot P \in \langle G_1 \rangle$ then

9: $S_0 := S_0 \cup \{\lambda \cdot P\}$

else if $-\lambda \cdot P \in \langle G_1 \rangle$ then

10: $S_1 := S_1 \cup \{\lambda \cdot P\}$

11: $J := S_0$

12: if $\exists e \in S_1$ then

13: for $e' \in S_1 \setminus \{e\}$ do

14: $q := e' \cdot e$

15: $J := J \cup \{q\}$

16: 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 $A \xrightarrow{\circ} B \xrightarrow{\circ}$ with $v, w$ reduced, $A \not= 0 \lor B \not= 0$)

2: if $v = w$ and $A, B \not= 0$ then

3: return $B \cdot A^{-1}$

4: return None
minimum (LEXMIN) and the arguments of the minimum (ARGLEXMIN) in Alg. 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 App. A. Since in the lexicographic ordering, the scalar is least significant (App. A), the resulting Pauli LIM has the same Pauli string as the the minimal Pauli LIM including scalar from App. A. Since in the lexicographic ordering, the scalar is least significant (App. A), the modulo scalar can easily be determined using the scalar-ignoring DivisionRemainder algorithm.

**Lemma 8.** Let \( v_0 \) and \( v_1 \) be LIMDD nodes, \( R \) a Pauli string and \( \nu, \nu' \in \mathbb{C} \). Define \( G = \text{Stab}(v_0) \cup \text{Stab}(v_1) \). 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 \( \lambda g \) 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 \( g_0, h_0 \in \text{Stab}(v_0) \) and \( g_1, h_1 \in \text{Stab}(v_1) \). Combining these yields \( \lambda(-1)^x g_0 g_1 = (-1)^y h_0 h_1 \).

We recall that, if \( g \in \text{Stab}(\nu) \) is a stabilizer of any state, then \( g^2 = I \). Therefore, squaring both sides of the equation, we get \( \lambda^2 (g_0 g_1)^2 = (h_0 h_1)^2 \), so \( \lambda^2 I = I \), so \( \lambda = \pm 1 \).

The central procedure in Alg. 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. D.1: by conjugation with a suitably chosen unitary \( U \), it maps \( G_1 \) to \( \{Z_1, Z_2, \ldots, Z_{|G_1|}\} \). Analogously to our explanation of INTERSECTSTABILIZERGROUPS, the group generated by \( U G_1 U^\dagger \) contains precisely all Pauli LIMs which satisfy the following three properties: (i) the scalar is \( 1 \); (ii) its Pauli string has an \( I \) or \( Z \) at positions \( 1, 2, \ldots, |G_1| \); (iii) its Pauli string has an \( I \) at positions \( |G_1| + 1, \ldots, n \). Therefore, the target \( g \) only exists if there is a LIM in \( \langle U G_0 U^\dagger \rangle \) which (i) has scalar \( -1 \) and satisfies properties (ii) and (iii). To find such a \( g \), we put \( U G_0 U^\dagger \) 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 \( G \mapsto U G U^\dagger \) 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 \( O(n^3) \).

### E Measuring an arbitrary qubit

Alg. 18 allows one to measure a given qubit. Specifically, given a quantum state \( |e\rangle \) represented by a LIMDD edge \( e \), a qubit index \( k \) and an outcome \( b \in \{0, 1\} \), it computes the probability of observing \( |b\rangle \) when measuring the \( k \)-th significant qubit of \( |e\rangle \). The algorithm proceeds by traversing the LIMDD with root edge \( e \) at Line 7. Like Alg. 5, which measured the top qubit, this algorithm finds the probability of a given outcome by computing the squared norm of the state when the \( k \)-th qubit is projected onto \( |0\rangle \), or \( |1\rangle \). The case that is added, relative to Alg. 5, is the case when \( n > k \), in which case it calls the procedure SQUAREDNORMPROJECTED. On input \( e, y, k \), the procedure SQUAREDNORMPROJECTED outputs the squared norm of \( \Pi_k^y |e\rangle \), where \( \Pi_k^y = I_{n-k} \otimes |y\rangle \langle y| \otimes I_{k-1} \) is the projector which projects the \( k \)-th qubit onto \( |y\rangle \).

After measurement of a qubit \( k \), a quantum state is typically projected to \( |0\rangle \) or \( |1\rangle \) \( (b = 0 \) or \( b = 1 \) \) on that qubit, depending on the outcome. Alg. 19 realizes this. It does so by traversing the LIMDD until a node \( v \) with \( \text{idx}(e) = k \) is reached. It then returns an edge to a new node by calling MAKEEDGE(\( \text{follow}_0(e) \), 0) to project onto \( |0\rangle \) or MAKEEDGE(0, \( \text{follow}_1(e) \)) to project onto \( |1\rangle \).
Algorithm 17 Algorithms $\text{LexMin}$ and $\text{ArgLexMin}$ 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 $\text{FindOpposite}$ for finding an element $g \in \langle G_0 \rangle$ such that $-g \in \langle G_1 \rangle$. A canonical choice for the $\text{RootLabel}$ (see Sec. 3.3.3) of an edge $e$ pointing to a node $v$ is $\text{LexMin}(G, \{I\}, \text{label}(e))$ where $G$ is a stabilizer generator group of $\text{Stab}(v)$.

\begin{algorithm}
\caption{Algorithm 17}
\begin{algorithmic}[1]
\State \textbf{output:} $\min_{(g_0, g_1) \in \langle G_0 \rangle \cdot \langle G_1 \rangle} A \cdot g_0 \cdot g_1$
\State \textbf{procedure} $\text{LexMin}$(stabilizer subgroup generating sets $G_0, G_1$ and Pauli LIM $A$)
\State \hspace{1em} $(g_0, g_1) := \text{ArgLexMin}(G_0, G_1, A)$
\State \hspace{1em} \textbf{return} $A \cdot g_0 \cdot g_1$
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{Algorithm 17}
\begin{algorithmic}[1]
\State \textbf{output:} $\arg \min_{g_0 \in G_0, g_1 \in G_1} A \cdot g_0 \cdot g_1$
\State \textbf{procedure} $\text{ArgLexMin}$(stabilizer subgroup generating sets $G_0, G_1$ and Pauli LIM $A$)
\State \hspace{1em} $(g_0, g_1) := \arg \min_{(g_0, g_1) \in \langle G_0 \rangle \cdot \langle G_1 \rangle} \{h \mid h \propto A \cdot g_0 \cdot g_1\}$
\State \hspace{1em} $\triangleright$ Using the scalar-ignoring DivisionRemainder algorithm from App. A,
\State \hspace{1em} $g' := \text{FindOpposite}(G_0, G_1, g_0, g_1)$
\State \hspace{1em} if $g'$ is None then
\State \hspace{2em} \textbf{return} $(g_0, g_1)$
\State \hspace{1em} else
\State \hspace{2em} $h_0, h_1 := g_0 \cdot g', (-g') \cdot g_1$
\State \hspace{2em} $\triangleright g_0g_1 = -h_0h_1$
\State \hspace{2em} if $A \cdot h_0 \cdot h_1 \prec_{\text{lex}} A \cdot g_0 \cdot g_1$ then \textbf{return} $(h_0, h_1)$
\State \hspace{2em} else \textbf{return} $(g_0, g_1)$
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{Algorithm 17}
\begin{algorithmic}[1]
\State \textbf{return} $g \in G_0$ such that $-g \in G_1$, or None if no such $g$ exists
\State \textbf{procedure} $\text{FindOpposite}$(stabilizer subgroup generating sets $G_0, G_1$)
\State \hspace{1em} Compute $U$ s.t. $UG_1U^\dagger = \{Z_1, Z_2, \ldots, Z_{|G_1|}\}$, using Algorithm 2 from \cite{82}
\State \hspace{1em} $\triangleright$ $Z_j$ is the $Z$ gate applied to qubit with index $j$
\State \hspace{1em} $H_0 := UG_0U^\dagger$
\State \hspace{1em} $H_0^{\text{RREF}} := H_0$ in RREF form
\State \hspace{1em} \textbf{for} $h \in H_0^{\text{RREF}}$ \textbf{do}
\State \hspace{2em} if $h$ satisfies all three of the following: (i) $h$ has scalar $-1$; the Pauli string of $h$ (ii) contains only $I$ or $Z$ at positions $1, 2, \ldots, |G_1|$, and (iii) only $I$ at positions $|G_1| + 1, \ldots, n$ \textbf{then}
\State \hspace{3em} \textbf{return} $U^\dagger hU$
\State \hspace{1em} return None
\end{algorithmic}
\end{algorithm}
Algorithm 18 Compute the probability of observing $|y⟩$ when measuring the $k$-th qubit of the state $|e⟩$. Here $e$ is given as LIMDD on $n$ qubits, $y$ is given as a bit, and $k$ is an integer index. For example, to measure the top-most qubit, one calls MEASURE($e, 0, n$). The procedure SQUAREDDNM($e, y, k$) computes the scalar $⟨e| (I ⊗ |y⟩⟨y| ⊗ I) |e⟩$, i.e., computes the squared norm of the state $|e⟩$ after the $k$-th qubit is projected to $|y⟩$. For readability, we omit calls to the cache, which implement dynamic programming.

1: 
2: procedure MEASUREMENTPROBABILITY(Edge $e$ $\overset{AP_n ⊗ P'}{→}$, $y ∈ \{0, 1\}$, $k ∈ [1...idx(e)]$)
3: if $n = k$ then
4: $p_0 := $ SQUAREDDNM(follow$_0(e)$)
5: $p_1 := $ SQUAREDDNM(follow$_1(e)$)
6: return $p_j/(p_0 + p_1)$ where $j = 0$ if $P_n ∈ \{I, Z\}$ and $j = 1$ if $P_n ∈ \{X, Y\}$
7: else if $v$ $\in$ SQUAREDDNMPROJECTED(follow$_0(e), y, k$)
8: $p_1 := $ SQUAREDDNMPROJECTED(follow$_1(e), y, k$)
9: return $(p_0 + p_1)/SQUAREDDNM(e)$
10: procedure SQUAREDDNM(Edge $e$ $\overset{AP_n ⊗ P'}{→}$)
11: if $n = 0$ then return $|λ|^2$
12: $s := ADD(SQUAREDDNM(follow$_0(→X)$), SQUAREDDNM(follow$_1(→Y)$))$
13: return $|λ|^2s$
14: procedure SQUAREDDNMPROJECTED(Edge $e$ $\overset{AP_n ⊗ P'}{→}$, $y ∈ \{0, 1\}$, $k ∈ [1...idx(e)]$)
15: $b := ($ $P_n ∈ \{X, Y\}$) $\triangleright$ i.e., $b = 1$ iff $P_n$ is Anti-diagonal
16: if $n = 0$ then
17: return $|λ|^2$
18: else if $n = k$ then
19: return SQUAREDDNM(follow$_b⊕y(e)$)
20: else
21: $α_0 := $ SQUAREDDNMPROJECTED(follow$_0(→X), b ⊕ y, k$)
22: $α_1 := $ SQUAREDDNMPROJECTED(follow$_1(→Y), b ⊕ y, k$)
23: return $|λ|^2 · (α_0 + α_1)$

on Line 6, recreating a node on level $k$ in the backtrack on Line 8. The projection operator $Π_k^b$ commutes with any LIM $P$ when $P_k$ is a diagonal operator (i.e., $P_k ∈ \{I_2, Z\}$). Otherwise, if $P_k$ is an anti-diagonal operator (i.e., $P_k ∈ \{X, Y\}$) have $Π_k^b · P = PΠ_k^{(1-b)}$. The algorithm applies this correction on Line 2. The resulting state should still be normalized as shown in Sec. 3.3.1.

F LIMDDs prepare the $W$ state efficiently

In this section, we prove Th. 8. To this end, we show that LIMDDs can efficiently simulate a circuit family given by McClung [56], which prepares the $|W⟩$ state when initialized to the $|0⟩$ state. We thereby show a separation between LIMDD and the Clifford+$T$ simulator, as explained in Sec. 3.4.3. Figure 16 shows the circuit for the case of 8 qubits.

Theorem 8. There exists a circuit family $C_n$ such that $C_n |0⟩^{⊗n} = |W_n⟩$, 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.

Proof. The proof outline is as follows. First, we establish that the LIMDD of each intermediate state
Algorithm 19 Project the state given by \( \text{LIMDD} \overset{A}{\rightarrow} \) to state \( |b\rangle \) for qubit \( k \), i.e., produce a LIMDD representing the state \( (\mathbb{I}_{n-k} \otimes |b\rangle \otimes \mathbb{I}_{k-1}) \cdot A |v\rangle \), with \( A = \lambda P_n \otimes \cdots \otimes P_1 \).

1: procedure UpdatePostMeas\((\text{Edge} \overset{\lambda P_n \otimes \cdots \otimes P_1}{\rightarrow}, k \in [1 \ldots \text{idx}(v)], b \in \{0, 1\})\)
2: \( b' := x \oplus b \) where \( x = 0 \) if \( P_k \in \{I, Z\} \) and \( x = 1 \) if \( P_k \in \{X, Y\} \) \( \triangleright \) flip \( b \) if \( P_k \) is anti-diagonal
3: if \( (v, k, b') \in \text{CACHE} \) then return \( \text{CACHE}[v, k, b'] \)
4: \( n := \text{idx}(v) \)
5: if \( n = k \) then
6: \( e := \text{MakeEdge}((1 - b') \cdot \text{low}_v, b' \cdot \text{high}_v) \) \( \triangleright \) Project \( |v\rangle \) to \( |b'\rangle \langle b'| \otimes I_{(n-k)}^2 \)
7: else \( \triangleright n \neq k: \)
8: \( e := \text{MakeEdge}(\text{UpdatePostMeas}(\text{low}_v, k, b'), \text{UpdatePostMeas}(\text{high}_v, k, b')) \)
9: \( \text{CACHE}[v, k, b'] := e \)
10: return \( e \)

(Lemma 11), as well as of each gate (Lemma 12), has polynomial size. Second, we establish that the algorithms presented in Sec. 3.3 can apply each gate to the intermediate state in polynomial time (Lemma 16). To this end, we observe that the circuit only produces relatively simple intermediate states. Specifically, each intermediate and output state is of the form \( |\psi_t\rangle = \frac{1}{\sqrt{n}} \sum_{k=1}^{n} |x_k\rangle \) where the \( x_k \in \{0, 1\}^n \) are computational basis vectors (Lemma 10). For example, the output state has \( |x_k\rangle = |0\rangle^{k-1} |1\rangle^{n-k} \). The main technical tool we will use to reason about the size of the LIMDDs of these intermediate states, are the subfunction rank and computational basis rank of a state. Both these measures are upper bounds of the size of a LIMDD (in Lemma 9), and also allow us to upper bound the time taken by the \text{APPLYGATE} and \text{ADD} algorithms (in Lemma 13 for \text{APPLYGATE} and Lemma 14 \text{ADD}).

The theorem follows from Lemma 16 and Corollary 16.

Fig. 16 shows the circuit for the case of \( n = 8 \) qubits. For convenience and without loss of generality, we only treat the case when the number of qubits is a power of 2, since the circuit is simplest in that case. In general, the circuit works as follows. The qubits are divided into two registers; register \( A \), with \( \log n \) qubits, and register \( B \), with the remaining \( n - \log n \) qubits. First, the circuit applies a Hadamard gate to each qubit in register \( A \), to bring the state to the superposition \( |+\rangle^{\otimes \log n} |0\rangle^{n-\log n} \). Then it applies \( n - \log n \) Controlled-X gates, where, in each gate, each qubit

![Figure 16: Reproduced from McClung [56]. A circuit on eight qubits (n = 8) which takes as input the |0⟩⊗⁸ state and outputs the |W⟩ state. In the general case, it contains log n Hadamard gates, and its Controlled-X gates act on one target qubit and at most log n control qubits.](image-url)
of register $A$ acts as the control qubits and one qubit in register $B$ is the target qubit. Lastly, it applies $n - \log n$ Controlled-X gates, where, in each gate, one qubit in register $B$ is the control qubit and one or more qubits in register $A$ are the target qubits. Each of the three groups of gates is highlighted in a dashed rectangle in Fig. 16. On input $|0\rangle^\otimes n$, the circuit’s final state is $|W_n\rangle$. We emphasize that the Controlled-X gates are permutation gates (i.e., their matrices are permutation matrices). Therefore, these gates do not influence the number of non-zero computational basis state amplitudes of the intermediate states. We refer to the $t$-th gate of this circuit as $U_t$, and the $t$-th intermediate state as $|\psi_t\rangle$, so that $|\psi_{t+1}\rangle = U_t |\psi_t\rangle$ and $|\psi_0\rangle = |0\rangle$ is the initial state.

We refer to the number of computational basis states with nonzero amplitude as a state’s computational basis rank, denoted $\chi_{\text{comp}}(|\psi\rangle)$.

**Definition 8.** (Computational basis rank) Let $|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha(x) |x\rangle$ be a quantum state defined by the amplitude function $\alpha : \{0,1\}^n \to \mathbb{C}$. Then the computational basis rank of $|\psi\rangle$ is $\chi_{\text{comp}}(|\psi\rangle) = |\{ x : \alpha(x) \neq 0 \}|$, the number of nonzero computational basis amplitudes.

Recall that, for a given function $\alpha : \{0,1\}^n \to \mathbb{C}$, a string $a \in \{0,1\}^\ell$ induces a subfunction $\alpha_u : \{0,1\}^{n-\ell} \to \mathbb{C}$, defined as $\alpha_u(y) = \alpha(y,x)$. We refer to the number of subfunctions of a state’s amplitude function as its subfunction rank. The following definition makes this more precise.

**Definition 9.** (Subfunction rank) Let $|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha^\psi(x) |x\rangle$ be a quantum state defined by the amplitude function $\alpha^\psi : \{0,1\}^n \to \mathbb{C}$, as above. Let $\chi_{\text{sub}}(|\psi\rangle, \ell)$ be the number of unique non-zero subfunctions induced by strings of length $\ell$, as follows,

$$\chi_{\text{sub}}(|\psi\rangle, \ell) = |\{ \alpha^\psi_y : \{0,1\}^{n-\ell} \to \mathbb{C} \mid \alpha^\psi_y \neq 0, y \in \{0,1\}^\ell \}| \quad (25)$$

We define the subfunction rank of $|\psi\rangle$ as $\chi_{\text{sub}}(|\psi\rangle) = \max_{\ell=0,\ldots,n} \chi_{\text{sub}}(|\psi\rangle, \ell)$. We extend these definitions in the natural way for an $n$-qubit matrix $U = \sum_{r,c \in \{0,1\}^n} \alpha^U(r,c) |r\rangle \langle c|$ defined by the function $\alpha^U : \{0,1\}^{2n} \to \mathbb{C}$.

It is easy to check that $\chi_{\text{sub}}(|\psi\rangle) \leq \chi_{\text{comp}}(|\psi\rangle)$ holds for any state.

For the next lemma, we use the notion of a prefix of a LIMDD node. This lemma will serve as a tool which allows us to show that a LIMDD is small when its computational basis rank is low. We apply this tool to the intermediate states of the circuit in Lemma 11.

**Definition 10** (Prefix of a LIMDD node). For a given string $x \in \{0,1\}^\ell$, consider the path traversed by the subroutine $\text{follow}_x(\begin{array}{c} R \end{array})$, which starts at the diagram’s root edge and ends at a node $v$ on level $\ell$. We will say that $x$ is a prefix of the node $v$. We let $\text{Labels}(x)$ be the product of the LIMs on the edges of this path (i.e., including the root edge). The set of prefixes of a node $v$ is denoted $\text{pre}(v)$.

**Lemma 9.** If a LIMDD represents the state $|\varphi\rangle$, then its width at any given level (i.e., the number of nodes at that level) is at most $\chi_{\text{comp}}(|\varphi\rangle)$.

**Proof.** For notational convenience, let us number the levels so that the root node is on level 0, its children are on level 1, and so on, with the Leaf on level $n$ (contrary to Fig. 3). Let $r$ be the root node of the LIMDD, and $R$ the root edge’s label. By construction of a LIMDD, the state represented by the LIMDD can be expressed as follows, for any level $\ell \geq 0$,

$$R |r\rangle = \sum_{x \in \{0,1\}^\ell} |x\rangle \otimes \text{follow}_x(\begin{array}{c} R \end{array}) \quad (26)$$

Since $\begin{array}{c} R \end{array}$ is the root of our diagram, if $x$ is a prefix of $v$, then

$$\text{follow}_x(\begin{array}{c} R \end{array}) = \text{Labels}(x) \cdot |v\rangle \quad (27)$$
A string \( x \in \{0, 1\}^\ell \) can be a prefix of only one node; consequently, the prefix sets of two nodes on the same level are disjoint, i.e., \( \text{pre}(v_p) \cap \text{pre}(v_q) = \emptyset \) for \( p \neq q \). Moreover, each string \( x \) is a prefix of some node on level \( \ell \) (namely, simply the node at which the \( \text{FOLLOW}_x(\cdot) \) subroutine arrives). Say that the \( \ell \)-th level contains \( m \) nodes, \( v_1, \ldots, v_m \). Therefore, the sets \( \text{pre}(v_1), \ldots, \text{pre}(v_m) \) partition the set \( \{0, 1\}^\ell \). Therefore, by putting Eq. 27 and Eq. 26 together, we can express the root node’s state in terms of the nodes \( v_1, \ldots, v_m \) on level \( \ell \):

\[
R|\psi\rangle = \sum_{k=1}^{m} \sum_{x \in \text{pre}(v_k)} |x\rangle \otimes \text{FOLLOW}_x(\cdot)
\]

\[
= \sum_{k=1}^{m} \sum_{x \in \text{pre}(v_k)} |x\rangle \otimes \text{Labels}(x) \cdot |v_k\rangle
\]

We now show that each term \( \sum_{x \in \text{pre}(v_k)} |x\rangle \otimes \text{Labels}(x) \cdot |v_k\rangle \) contributes a non-zero vector. It then follows that the state has computational basis rank at least \( m \), since these terms are vectors with pairwise disjoint support, since the sets \( \text{pre}(v_k) \) are pairwise disjoint. Specifically, we show that each node has at least one prefix \( x \) such that \( \text{Labels}(x) \cdot |v\rangle \) is not the all-zero vector. In principle, this can fail in one of three ways: either \( v \) has no prefixes, or all prefixes \( x \in \text{pre}(v_k) \) have \( \text{Labels}(x) = 0 \) because the path contains an edge labeled with the 0 LIM, or the node \( v \) represents the all-zero vector (i.e., \( |v\rangle = 0 \)). First, we note that each node has at least one prefix, since each node is reachable from the root, as a LIMDD is a connected graph. Second, due to the zero edges rule (see Def. 5), for any node, at least one of its prefixes has only non-zero LIMs on the edges. Namely, each node \( v \) has at least one incoming edge labeled with a non-zero LIM, since, if it has an incoming edge from node \( w \) labeled with 0, then this must be the high edge of \( w \) and by the zero edges rule the low edge of \( w \) must also point to \( v \) and moreover must be labeled with \( 1 \) by the low factoring rule. Together, via a simple inductive argument, there must be at least one non-zero path from \( v \) to the root. Lastly, no node represents the all-zero vector, due to the low factoring rule (in Def. 5). Namely, if \( v \) is a node, then by the low factoring rule, the low edge has label 1. Therefore, if this edge points to node \( v_0 \), and the high edge is \( \rightarrow \), then the node \( v \) represents \( |v\rangle = |00\rangle + |11\rangle A |v_1\rangle \) with possibly \( A = 0 \), so, if \( |v_0\rangle \neq 0 \), then \( |v\rangle \neq 0 \). An argument by induction now shows that no node in the reduced LIMDD represents the all-zero vector.

Therefore, each node has at least one prefix \( x \) such that \( \text{FOLLOW}_x(\cdot) \neq 0 \). We conclude that the equation above contains at least \( m \) non-zero contributions. Hence \( m \leq \chi_{\text{comp}}(R|\psi\rangle) \), at any level \( 0 \leq \ell \leq n \).

\[\text{Lemma 10.}\] Each intermediate state in the circuit in Fig. 16 (with \( n = 2^c \)) has \( \chi_{\text{comp}}(|\psi\rangle) \leq n \).

\[\text{Proof.}\] The initial state is \( |\psi_0\rangle = |0\rangle^\otimes n \), which is a computational basis state, so \( \chi_{\text{comp}}(\psi_0) = 1 \). The first \( \log n \) gates are Hadamard gates, which produce the state

\[
|\psi_{\log n}\rangle = H^\otimes \log n \otimes 1^{n-\log n} |0\rangle^\otimes n \otimes 0^\otimes n-\log n = \frac{1}{\sqrt{n}} \sum_{x=0}^{n-1} |x\rangle |0\rangle^\otimes n^\otimes n-\log n
\]

This is a superposition of \( n \) computational basis states, so we have \( \chi_{\text{comp}}(|\psi_{\log n}\rangle) = n \). All subsequent gates are controlled-X gates; these gates permute the computational basis states, but they do not increase their number.

\[\text{Lemma 11.}\] The reduced LIMDD of each intermediate state in the circuit in Fig. 16 has polynomial size.

\[\text{Proof.}\] By Lemma 9, the width of a LIMDD representing \( |\varphi\rangle \) is at most \( \chi_{\text{comp}}(|\varphi\rangle) \) at any level. Since there are \( n \) levels, the total size is at most \( n \chi_{\text{comp}}(|\varphi\rangle) \). By Lemma 10, the intermediate states in question have polynomial \( \chi_{\text{comp}} \), so the result follows.

\[\text{Lemma 12.}\] The LIMDD of each gate in the circuit in Fig. 16 (with \( n = 2^c \)) has polynomial size.
Proof. Each gate acts on at most $k = \log n + 1$ qubits. Therefore, the width of any level of the LIMDD is at most $4^k = 4n^2$. The height of the LIMDD is $n$ by definition, so the LIMDD has at most $4n^3$ nodes. 

The \texttt{ApplyGate} procedure handles the Hadamard gates efficiently, since they apply a single-qubit gate to a product state. The difficult part is to show that the same holds for the controlled-X gates. To this end, we show a general result for the speed of LIMDD operations (Lemma 13). Although this worst-case upper bound is tight, it is exponentially far removed from the best case, e.g., in the case of Clifford circuits, in which case the intermediate states can have exponential \texttt{chsub}, yet the LIMDD simulation is polynomial-time, as shown in Sec. 3.3.4.

**Lemma 13.** The number of recursive calls made by subroutine \texttt{ApplyGate}(\(U, |\psi\rangle\)), is at most \(n\texttt{chsub}(U)\texttt{chsub}(|\psi\rangle)\), for any gate \(U\) and any state \(|\psi\rangle\).

Proof. Inspecting Alg. 8, we see that every call to \texttt{ApplyGate}(\(U, |\psi\rangle\)) produces four new recursive calls, namely \texttt{ApplyGate}(\texttt{follow}_{rc}(U), \texttt{follow}_{c}(|\psi\rangle)) for \(r, c \in \{0, 1\}\). Therefore, the set of parameters in all recursive calls of \texttt{ApplyGate}(\(U, |\psi\rangle\)) is precisely the set of tuples \((\texttt{follow}_{rc}(U), \texttt{follow}_{c}(|\psi\rangle))\), with \(r, c \in \{0, 1\}^\ell\) with \(\ell = 0 \ldots n\). The terms \texttt{follow}_{rc}(U) and \texttt{follow}_{c}(|\psi\rangle) are precisely the subfunctions of \(U\) and \(|\psi\rangle\), and since there are at most \(\texttt{chsub}(U)\) and \(\texttt{chsub}(|\psi\rangle)\) of these, the total number of distinct parameters passed to \texttt{ApplyGate} in recursive calls at level \(\ell\), is at most \(\texttt{chsub}(U, \ell) \cdot \texttt{chsub}(|\psi\rangle, \ell) \leq \texttt{chsub}(U) \cdot \texttt{chsub}(|\psi\rangle)\). Summing over the \(n\) levels of the diagram, we see that there are at most \(n\texttt{chsub}(U)\texttt{chsub}(|\psi\rangle)\) distinct recursive calls in total. As detailed in Sec. 3.3.3, the \texttt{ApplyGate} algorithm caches its inputs in such a way that it will achieve a cache hit on a call \texttt{ApplyGate}(\(U', |\psi'\rangle\)) when it has previously been called with parameters \(U, |\psi\rangle\) such that \(U = U'\) and \(|\psi\rangle = |\psi'\rangle\). Therefore, the total number of recursive calls that is made, is equal to the number of distinct calls, and the result follows.

In our case, both \(\texttt{chsub}(U)\) and \(\texttt{chsub}(|\psi\rangle)\) are polynomial, so a polynomial number of recursive calls to \texttt{ApplyGate} is made. We now show that also the ADD subroutine makes only a small number of recursive calls every time it is called from \texttt{ApplyGate}. First, Lemma 14 shows expresses a worst-case upper bound on the number of recursive calls to ADD in terms of \(\texttt{chsub}\). Then Lemma 15 uses this result to show that, in our circuit, the number of recursive calls is polynomial in \(n\).

**Lemma 14.** The number of recursive calls made by the subroutine ADD(\(|\alpha\rangle, |\beta\rangle\)) is at most \(n\texttt{chsub}(\langle\alpha|) \cdot \texttt{chsub}(\langle|\beta\rangle)\), if \(|\alpha\rangle, |\beta\rangle\) are \(n\)-qubit states.

Proof. Inspecting Alg. 9, every call to \texttt{ADD}(\(\langle|\alpha\rangle, |\beta\rangle\)) produces two new recursive calls, namely \texttt{ADD}(\texttt{follow}_{q0}(\langle|\alpha\rangle), \texttt{follow}_{q1}(\langle|\beta\rangle))\) and \texttt{ADD}(\texttt{follow}_{q1}(\langle|\alpha\rangle), \texttt{follow}_{q1}(\langle|\beta\rangle))\). It follows that the set of parameters on \(n - \ell\) qubits with which ADD is called is the set of tuples \((\texttt{follow}_{q0}(\langle|\alpha\rangle), \texttt{follow}_{q1}(\langle|\beta\rangle))\), for \(x \in \{0, 1\}^\ell\). This corresponds precisely to the set of subfunctions of \(\alpha\) and \(\beta\) induced by length-\(\ell\) strings, of which there are \(\texttt{chsub}(\langle|\alpha\rangle, \ell)\) and \(\texttt{chsub}(\langle|\beta\rangle, \ell)\), respectively. Because the results of previous computations are cached, as explained in Sec. 3.3.3, the total number of recursive calls is the number of distinct recursive calls. Therefore, we get the upper bound of \(\texttt{chsub}(\langle|\alpha\rangle) \cdot \texttt{chsub}(\langle|\beta\rangle)\) for each level of the LIMDD. Since the LIMDD has \(n\) levels, the upper bound \(n\texttt{chsub}(\langle|\alpha\rangle) \cdot \texttt{chsub}(\langle|\beta\rangle)\) follows.

**Lemma 15.** The calls to \texttt{ADD}(\(\langle|\alpha\rangle, |\beta\rangle\)) that are made by the recursive calls to \texttt{ApplyGate}(\(U_t, |\psi_t\rangle\)), satisfy \(\texttt{chsub}(\langle|\alpha\rangle), \texttt{chsub}(\langle|\beta\rangle) = \text{poly}(n)\).

Proof. We have established that the recursive calls to \texttt{ApplyGate} are all called with parameters of the form \texttt{ApplyGate}(\texttt{follow}_{r,c}(U_t), \texttt{follow}_{c}(|\psi_t\rangle)) for some \(r, c \in \{0, 1\}^\ell\). Inspecting Alg. 8, we see that, within such a call, each call to \texttt{ADD}(\(\langle|\alpha\rangle, |\beta\rangle\)) has parameters which are both of the form \(|\alpha\rangle, |\beta\rangle = \texttt{ApplyGate}(\texttt{follow}_{r,c,y}(U_t), \texttt{follow}_{c}(|\psi_t\rangle))\) for some \(x, y \in \{0, 1\}\); therefore, the parameters \(|\alpha\rangle, |\beta\rangle\) are of the form \(|\alpha\rangle, |\beta\rangle = \texttt{follow}_{r,c}(U_t) \cdot \texttt{follow}_{c}(|\psi_t\rangle)\). Here \(\text{follow}_{c}(|\psi_t\rangle)\) is a quantum state on \(n - (\ell + 1)\) qubits.
The computational basis rank of a state is clearly non-increasing under taking subfunctions; that is, for any string \( x \), it holds that, \( \chi_{\text{comp}}(\text{follow}_x(|\psi\rangle)) \leq \chi_{\text{comp}}(|\psi\rangle) \). In particular, we have \( \chi_{\text{comp}}(\text{follow}_{x,y}(|\psi\rangle)) \leq \chi_{\text{comp}}(|\psi\rangle) = \mathcal{O}(n) \). The matrix \( \text{follow}_{x,y}(U_t) \) is a subfunction of a permutation gate, and applying such a matrix to a vector cannot increase its computational basis rank, so we have

\[
\chi_{\text{sub}}(|\alpha\rangle) = \chi_{\text{sub}}(\text{follow}_{x,y}(U_t) \cdot \text{follow}_{cy}(|\psi\rangle)) \\
\leq \chi_{\text{comp}}(\text{follow}_{x,y}(U_t) \cdot \text{follow}_{cy}(|\psi\rangle)) \\
\leq \chi_{\text{comp}}(|\psi\rangle) = \mathcal{O}(n)
\]

(31)

(32)

(33)

This proves the lemma.

\( \square \)

Lemma 16. Each call to \textsf{ApplyGate}(U_t, |\psi_t\rangle) runs in polynomial time, for any gate \( U_t \) in the circuit in Fig. 16 (with \( n = 2^c \)).

Proof. If \( U_t \) is a Hadamard gate, then \textsf{LIMDD} can apply this in polynomial time by Th. 6, since \( |\psi_t\rangle \) is a stabilizer state. Otherwise, \( U_t \) is one of the controlled-X gates. In this case there is a polynomial number of recursive calls to \textsf{ApplyGate}, by Lemma 13. Each recursive call to \textsf{ApplyGate} makes two calls to \textsf{Add}(|\alpha\rangle, |\beta\rangle), where both \( \alpha \) and \( \beta \) are states with polynomial subfunction rank, by Lemma 15. By Lemma 14, these calls to \textsf{Add} all complete in time polynomial in the subfunction rank of its arguments.

\( \square \)

Corollary 16. The circuit in Fig. 16 (with \( n = 2^c \)) can be simulated by \textsf{LIMDD}s in polynomial time.

G 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-\textsf{LIMDD}s, it would be highly interesting to theoretically compare Pauli-\textsf{LIMDD}s 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-\textsf{LIMDD}s can efficiently simulate and using Bravyi et al.’s heuristic algorithm for searching the stabilizer rank of the circuits’ output states [12]. 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 \( |D_w^n\rangle \), which are equal superpositions of all \( n \)-qubit computational-basis status with Hamming weight \( w \) (note \( |D_w^n\rangle = |D_w^n\rangle \)),

\[
|D_w^n\rangle = \frac{1}{\sqrt{\binom{n}{w}}} \sum_{x:|x|=w} |x\rangle
\]

(34)

We implemented the algorithm by Bravyi et al.: see [83] 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 \textsf{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 [84]. Table 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.

Accepted in JQuantum 2023-07-17, click title to verify. Published under CC-BY 4.0.
Table 2: Heuristically-found upper bounds on the stabilizer rank $\chi$ of Dicke states $|D_n^w\rangle$ (eq. (34)) using the heuristic algorithm from Bravyi et al. [12], see text in App. G 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 $w > \left\lfloor \frac{n}{2} \right\rfloor$ since $|D_n^w\rangle$ and $|D_n^{n-w}\rangle$ have identical stabilizer rank because $X^\otimes n |D_n^w\rangle = |D_n^{n-w}\rangle$. For $|D_8^3\rangle$ and $|D_9^4\rangle$, 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.

<table>
<thead>
<tr>
<th>Hamming weight $w$</th>
<th>#qubits $n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>$\leq 11$</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>4</td>
<td>8</td>
<td>$&gt; 10$?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 decompositions 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. [12], which has been explained in more detail in [84]. The algorithm follows a simulated annealing approach: on input $n, w$ and $\chi$, it performs a random walk through sets of $\chi$ stabilizer states. It starts with a random set $V$ of $\chi$ 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 $F_V := |\langle D_n^w | \Pi_V | D_n^w \rangle|$ with $\Pi_V$ 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 $\beta$ should be interpreted as the inverse temperature. The algorithm terminates if it finds $F_V = 1$, implying that $|D_n^w\rangle$ can be written as linear combination of $V$, outputting the number $\chi$ as (an upper bound on) the stabilizer rank of $|\psi\rangle$. For a fixed $\chi$, we use identical values to Bravyi et al. [12] and vary $\beta$ from 1 to 4000 in 100 steps, performing 1000 steps at each value of $\beta$. 

Accepted in Quantum 2023-07-17, click title to verify. Published under CC-BY 4.0.