PauLie: Fast Classification of Pauli Dynamical Lie Algebras

Oxana Shaya*, Konstantin Golovkin†, Mainak Roy‡, Vincent Russo§
*Institute for Information Processing (tnt/L3S), Leibniz Universität Hannover, Germany
Email: shayaoxana@gmail.com
†Independent Researcher, Russia
‡Tata Institute of Fundamental Research, Mumbai, India
§Unitary Foundation, USA

***Abstract*—The dynamical Lie algebra (DLA) governs the controllability, expressibility, and simulation complexity of a quantum system. Explicitly computing it has been a major computational bottleneck: brute-force Lie closure scales exponentially in the number of qubits n. Many applications, however, consult only the isomorphism type of the DLA. We introduce PauLie, an open-source framework that decides this isomorphism type for DLAs generated by arbitrary Pauli strings, building on the anticommutation-graph reduction of Aguilar et al. PauLie runs in $O(n|\mathcal{G}|\max(n,|\mathcal{G}|))$ time, where $|\mathcal{G}|$ is the number of generators, turning DLA classification into a routine preprocessing step. We demonstrate its use as a structural oracle for routing Lie-algebraic simulation and Cartan decomposition, diagnosing barren plateaus in variational quantum algorithms, and engineering universal Pauli string generator sets with optimal generation rate.**

***Index Terms*—Dynamical Lie algebra, quantum control, quantum software, Hamiltonian simulation, variational quantum algorithms**

I. Introduction

As quantum devices scale, efficient tools for compilation and controllability analysis become increasingly important. The continuous-time evolution of a quantum system traces a trajectory in Hilbert space whose reachable set is determined by the underlying control Hamiltonian [1, 2, 3]. For finite-dimensional closed quantum systems with piecewise-continuous controls, the set of reachable unitary operations is associated with a connected Lie group generated by the control Hamiltonian. The tangent space to this group, the real span of the Hamiltonian terms and their nested commutators, is the dynamical Lie algebra (DLA). The DLA's structure dictates a system's symmetries, expressibility, and controllability. Explicitly determining a system's DLA has been a severe computational bottleneck. Brute-force methods build a basis by closing the generators under nested commutators, and the dimension of that basis is generically exponential in n: for a universal generator set it is $4^n - 1$. Any method that returns a basis therefore costs exponential time and memory on such systems, however the closure is organized. Recently there have been several breakthroughs. The DLAs of 2-local spin systems have been classified [4, 5]. This line of research culminated in a classification of DLAs generated by arbitrary sets of Pauli strings [6].

PauLie [7] turns this classification into a general-purpose software framework built around an efficient graph-theoretic reduction algorithm. The algorithm achieves $O(n|\mathcal{G}|\max(n,|\mathcal{G}|))$ scaling with respect to the number of qubits n and number of generators $|\mathcal{G}|$, and improves upon the concurrently proposed algorithm [8] whenever $|\mathcal{G}| \neq n$. PauLie thus makes DLA analysis routinely available to quantum information scientists and engineers, who can use it in several ways: First, as a key ingredient for Hamiltonian simulation and circuit synthesis [9]. Second, to identify the scope of efficient classical algorithms to simulate quantum systems with a small DLA [10, 11]. Third, to infer expressibility and barren-plateau behavior in algorithms like QAOA [12, 13, 14]. The framework can be used to study how the DLA changes by changing the generator set [15] and therefore has relevance in quantum system engineering. We demonstrate this by concretely identifying universal generator sets with optimal generation rate. More generally, graph-based representations of commutation relations between quantum observables appear in several areas of quantum information theory. For example, the anticommutation graph formalism provides a graph-theoretic description of qudit observable families [16]. Moreover, we extended our framework to capture properties of the commutator graph which are related to quantities in quantum chaos and complexity [17].

II. Technical Background

To understand PauLie's classification algorithm, we formalize the relationship between quantum dynamics, Lie algebras, and graph-theoretic representations.

A. Quantum Dynamics and Matrix Lie Algebras

A state of an isolated physical system is represented at a fixed time t by a vector $|\psi\rangle$ in a Hilbert space $\mathcal{H}$, that is, a complex vector space endowed with an inner product. The time evolution of the state is determined by the Schrödinger equation $i\frac{d}{dt}|\psi\rangle = H|\psi\rangle$, where we set $\hbar = 1$. H is the Hamiltonian that describes the observable corresponding to the total energy of the system. An observable is a Hermitian linear map acting on $\mathcal{H}$. The solution to this differential equation with initial state $|\psi_0\rangle$ and time-independent Hamiltonian H is $|\psi(t)\rangle = e^{-iHt}|\psi_0\rangle$. For a Hamiltonian that decomposes as

a sum $H = \sum_j h_j$, the terms h_j typically do not commute, so e^{-iHt} does not factor directly into a product of single-term exponentials. The Lie-Trotter formula resolves this by approximating

$$e^{-iHt} = \lim_{m\to\infty} \Big(\prod_j e^{-ih_j t/m}\Big)^m.$$

We specify a set of Hermitian operators $\mathcal{G} = \{h_j\}_{j\in J}$ that describe the ways in which we can control the quantum system. Depending on the context, the h_j may be interaction terms of a Hamiltonian one wishes to simulate [18, 19], control Hamiltonians of a quantum device [20, 21], or generators of a variational ansatz [22]. The reachable states are then of the form $|\psi(\vec{\theta})\rangle = e^{-ih_{l_m}\theta_m}\dots e^{-ih_{l_1}\theta_1}|\psi_0\rangle$ for some index sequence $(l_1,\dots,l_m) \in J^m$ and parameters $\vec{\theta} \in \mathbb{R}^m$, which may be interpreted as evolution times, control amplitudes, or variational angles. The matrix Lie algebra $\mathfrak{g}$ associated with $\mathcal{G}$, that is, the smallest real subspace of anti-Hermitian matrices containing $\{ih_j\}_{j\in J}$ and closed under the matrix commutator $[h,g] = hg - gh$, is called the dynamical Lie algebra (DLA) [23, 24]. By the Baker-Campbell-Hausdorff formula,

$$e^A e^B = \exp\Big(A + B + \tfrac{1}{2}[A,B] + \tfrac{1}{12}[A,[A,B]] - \tfrac{1}{12}[B,[A,B]] + \cdots\Big),$$

the elements in the Lie algebra generate precisely the unitaries that are reachable [25]. The classical compact simple Lie algebras are:

$$\mathfrak{su}(d) = \{x \in \mathbb{C}^{d\times d} \mid x = -x^\dagger, \mathrm{tr}(x) = 0\}$$
$$\mathfrak{so}(d) = \{x \in \mathbb{R}^{d\times d} \mid x = -x^T\}$$
$$\mathfrak{sp}(d) = \{x \in \mathfrak{su}(2d) \mid x = -\Omega x^T \Omega^T\}$$

where the symplectic form is denoted as $\Omega = \begin{pmatrix} 0 & \mathbb{I}_d \\ -\mathbb{I}_d & 0 \end{pmatrix}$. For a system of n qubits, the Hilbert space is $\mathcal{H} = \mathbb{C}^{2^n}$. A generator set whose DLA equals $\mathfrak{su}(2^n)$ is called universal, meaning the system is fully controllable.

B. Graph-Theoretic Mapping of Lie Algebras

PauLie avoids dense matrices by mapping the generation of Pauli Lie algebras to a graph reduction problem [6]. The algebraic structure is extracted by analyzing the anticommutation relations of the generators.

Definition 1 (Anticommutation Graph)**.** *The anticommutation graph $\Gamma = (\mathcal{G}, E)$ has the generating set $\mathcal{G}$ as its vertex set. Undirected edges E connect vertices whose corresponding Pauli operators anti-commute: $E = \{(P,Q) \mid \{P,Q\} = 0\}$.*

Any connected anticommutation graph can be reduced to a canonical representative by operations that alter the graph topology but leave the resulting DLA invariant.

Definition 2 (Lightning)**.** *The lightning of an anticommutation graph with respect to a vertex V is the induced subgraph of $\mathcal{G}\setminus\{V\}$ with binary labeling of each vertex $W \neq V$ as either* lit *if $\{V,W\} = 0$ or else* unlit*.*

Definition 3 (Contraction)**.** *A contraction along an edge (V,W) replaces the generator V with the scaled commutator $X \propto [V,W]$.*

Structurally, X remains adjacent to W, while every other vertex toggles its adjacency to the contracted vertex precisely if it is lit in the lightning with respect to W: common neighbours of V and W lose their edge, and vertices adjacent to W alone gain one. We illustrate a contraction in Figure 1.

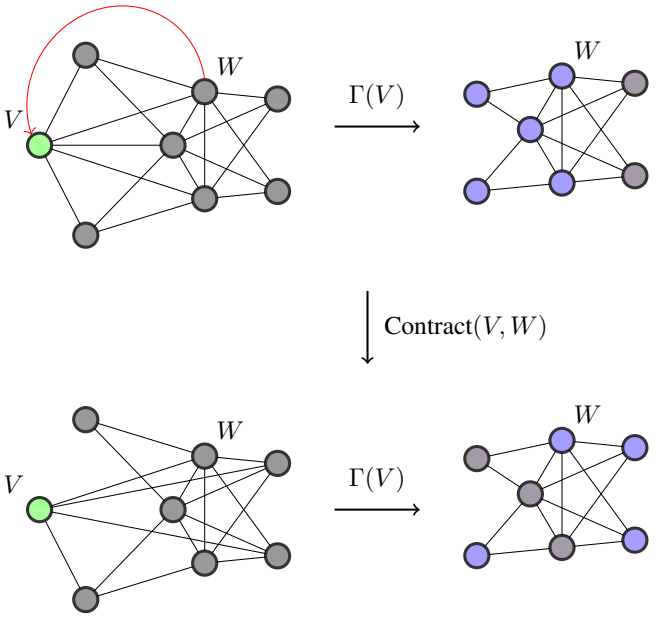

Figure 1. The anticommutation graph of $\mathcal{G} = \{YXIZ,\ ZZII,\ XIII,\ ZIII,\ YIIX,\ YIIZ,\ YYIX,\ YYXX\}$ is drawn in the upper left. Its lightning $\Gamma(V)$ is drawn in the upper right where the blue marks lit vertices. The result of contracting $V = YIIX$ by $W = ZIII$ to $XIIX$ is drawn in the lower left. The lightning after the contraction is shown in the lower right.

By systematically applying contractions the graph is mapped to a canonical structure: a line graph (Type A) or a *starlike* graph (Type B). For Type A graphs, n_L denotes the number of vertices in the line and n_c the number of single vertices attached to the penultimate vertex. A starlike graph consists of a central vertex (the core) connected to a set of disjoint simple path graphs (legs). If the graph has no legs longer than 2, it is of Type B1. If it has exactly one leg of length 4 and none of length 3, it is Type B2; otherwise it is Type B3. For starlike graphs $n_c + 1$ denotes the number of legs of length 1, and n_2 the number of legs of length 2.

An example of such a canonical graph is presented in Figure 2.

III. Pauli DLA Classification Algorithm

PauLie implements the classification of Aguilar et al. [6] as an optimized graph algorithm. The algorithm operates directly

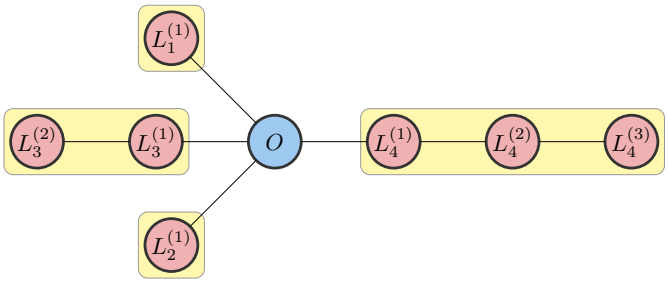

Figure 2. An example of a canonical starlike graph of type B3. The central vertex is O and leg nodes are denoted by $L_i^{(j)}$ where i counts the leg and j counts the node inside the leg. Here $n_c = 1$, $n_2 = 1$ and the associated dynamical Lie algebra is $\mathfrak{su}(8) \oplus \mathfrak{su}(8)$.

on the anticommutation graph of the generator set. Unlike methods that rely on invariants of an associated quadratic space [8], it achieves $O(n|\mathcal{G}|\max(n, |\mathcal{G}|))$ scaling through graph transformations. The algorithm assumes no locality; its input is an arbitrary set of Pauli strings.

By [6, Section VI], the connected components of the anticommutation graph correspond to the direct sum of the Lie algebras generated by each connected component. Therefore, PauLie first isolates these components and independently canonicalizes each component. The algorithm iteratively builds the canonicalized graph by adding vertices in a loop and utilizing contractions to ensure the canonical property is maintained in every iteration. In the following description, we count the number of *basic operations*, by which we mean multiplication and commutation of two Pauli strings.

a) Canonicalization Routine: Vertices are processed from a stack, initialized with a DFS preorder of the component. The routine maintains a single invariant: after every iteration, the vertices processed so far form a canonical graph—a central vertex with legs sorted by length—that generates the same DLA as the generators it was built from. We describe an iteration from the perspective of the vertex V being added; lit and unlit refer to the lightning with respect to V (Definition 2), and we call the longest leg the *long leg*. Contractions (Definition 3) are the basic move; occasionally the routine instead multiplies a vertex already in the graph by other generators, which likewise preserves the DLA. Each iteration reduces the lightning until V is adjacent only to vertices at which the canonical form admits an attachment, and attaches it there:

1) **Building the core:** The first vertex becomes the central vertex. While the graph has fewer than two legs, each added vertex is attached directly to the centre: if it is lit on the existing leg, contractions with that leg and with the centre first remove the unwanted adjacency, and V then forms a new leg of length 1.
2) **Handling legs of length 1:** All legs of length 1 must be in the same state, either all lit or all unlit, so that one of them can act as a *representative* for the rest. If instead some leg P is lit and some leg Q is unlit, we apply the reduction behind the first part of [6, Theorem 4]: multiplying every other lit vertex by the product PQ preserves the generated algebra and turns its light off, so V ends up adjacent to P alone, attaches there, and the iteration ends with a new leg of length 2. If this gives a type-A graph a second leg of length at least 2, the graph becomes type B; since type B admits no leg longer than 4, the long leg is trimmed to length 4, and the removed vertices are pushed back onto the stack to be reprocessed later (a *re-add*). When the length-1 legs already agree, contractions make V unlit on the representative—hence on all of them—and lit on the central vertex. If no vertex of any longer leg is lit either, V simply becomes a new leg of length 1 and the iteration ends.
3) **Transferring the lightning to the long leg:** Next, the remaining light is concentrated on the long leg. Each length-2 leg with a lit vertex is cleared by contractions with its own vertices, the first vertices of the long leg, and the representative; in exchange, the light moves onto the long leg. If after this only the central vertex is lit, V attaches to it as a new leg of length 1 and the iteration ends.
4) **Handling the long leg:** Only the central vertex and the long leg now carry light. If the central vertex is unlit, contracting V successively with the long-leg vertices, from the first lit position down to the first vertex, lights it. (One shortcut comes first: in a type-A graph whose long leg has length at least 5, with only the last vertex lit, V is appended at the end of the leg directly.) While at least two long-leg vertices remain lit, an inner loop contracts V with contiguous runs of the leg; every pass moves the lit positions towards the end of the leg, so eventually exactly one vertex is lit. Precomputed prefix products make each of these contractions cost $O(1)$ basic operations, as exploited in the runtime analysis below. Two cases remain.
    a) **First vertex is lit:** Contracting along the whole leg moves the light to its end, and V is appended there. The exception is type B2, whose long leg must not grow: there, the move of [6, Lemma 3] attaches V to the central vertex as a new leg of length 1 instead.
    b) **Some other vertex is lit:** The long leg is broken just before the lit position: the preceding vertex absorbs V and the representative, and the tail of the leg, now headed by V, becomes a second leg attached to the centre. If the graph was type A and is left with two legs of length at least 2, it becomes type B, and the surplus is trimmed—the longer leg to length 4, the other to length 2—with all removed vertices re-added via the stack. A re-add is not an actual removal; it is a device to generate, over the next iterations, the contraction sequence that returns the graph to canonical form.

At the end of the procedure, we must remove all dependent Pauli strings for the correct classification. By [6, Section IV.A],

we only need to perform Gaussian elimination over the legs of length 1.

Algorithm 1 PauLie Canonicalizer

Require: Pauli generator set $\mathcal{G}$
$S \leftarrow$ Vertex stack
$C \leftarrow$ **null** ▷ Central vertex
$L \leftarrow []$ ▷ List of graph legs
while S is not empty **do**
  $V \leftarrow \text{pop}(S)$
  if C is **null then**
    $C \leftarrow V$
    continue
  else if length$(L) < 2$ **then**
    Build core by contracting V into C and L_1
    continue
  else if length-1 legs in L have differing lit states **then**
    Use [6, Theorem 4] to create a length-2 leg
    while length of longest leg in $L > 4$ **do**
      Pop last vertex from longest leg in L
      Push vertex to S
    end while
    continue
  end if
  $\omega \leftarrow$ any length-1 leg
  Make ω unlit and C lit
  if only C is lit **then**
    Add V to L as a length-1 leg
    continue
  end if
  if any length-2 leg in L has a lit vertex **then**
    Transfer lightning away from length-2 legs
  end if
  Reduce lightning on the longest leg
end while
Perform Gaussian elimination over length-1 legs in L
return Canonical graph constructed from C and L

The specific geometric configuration of the resulting canonical graph uniquely identifies the DLA as one of four families, as detailed in Table I. There is one exception, which is a single Pauli string. This case generates the algebra $\mathfrak{u}(1)$.

Type	Algebraic Isomorphism
A	$\bigoplus_{i=1}^{2^{n_c}} \mathfrak{so}(n_L+1)$
B1	$\bigoplus_{i=1}^{2^{n_c}} \mathfrak{sp}(2^{n_2})$
B2	$\bigoplus_{i=1}^{2^{n_c}} \mathfrak{so}(2^{n_2+3})$
B3	$\bigoplus_{i=1}^{2^{n_c}} \mathfrak{su}(2^{n_2+2})$

Table I
CORRESPONDENCE BETWEEN THE CANONICAL GRAPH TYPE AND THE ASSOCIATED DLA. TYPE A CORRESPONDS TO LINE GRAPHS, WHILE TYPE B CORRESPONDS TO STARLIKE GRAPHS.

b) Runtime Analysis: Comparison to prior work. Standard numerical Lie-closure algorithms perform rank checks on a $2^{2n} \times N$ basis matrix, where N is the current closure dimension; even with recent optimizations [26] they remain exponential in the worst case. The Lie closure is sufficient but not necessary to determine the isomorphism type. The concurrent algebraic approach of [8] does decide the isomorphism type through $\mathbb{F}_2$ quadratic-space invariants in $\mathcal{O}(\max(n, |\mathcal{G}|)^3)$ time. PauLie improves on that bound whenever $|\mathcal{G}| \neq n$ and matches it at $|\mathcal{G}| = n$, as we show below.

Cost model. Under PauLie's bit-packed symplectic encoding, a Pauli string on n qubits occupies $O(n/W)$ words, where W is the word size (typically 64). Multiplication and commutation of two Pauli strings reduce to bitwise operations on these arrays and cost $O(n/W)$ each.

Cost per iteration. The *main loop* of Algorithm 1 is its outer **while** loop, and one execution of its body—popping a single vertex from the stack and processing it through Steps 1–4 of the canonicalization routine—is an *iteration*. The cost of a single iteration breaks down as:

Step	1	2	3	4
Cost (basic operations)	$O(1)$	$O(\|\mathcal{G}\|)$	$O(\|\mathcal{G}\|)$	$O(\|\mathcal{G}\|)$

The optimization in Step 4 relies on three facts: contractions reduce to multiplications when the result is known to be non-zero; Pauli strings are involutory; and coefficients are immaterial for the classification. Together these let prefix products along the long leg be precomputed once per iteration in $O(|\mathcal{G}|)$ time. Since a lit-vertex reduction corresponds to contraction with a contiguous range of vertices, the earlier facts mean we can compute this by just taking the product of two precomputed prefix products. Thus each of the $O(|\mathcal{G}|)$ lit-vertex reductions costs $O(1)$. Per-iteration total: $O(|\mathcal{G}|)$ basic operations.

Main-loop total. The number of iterations equals the number of vertices ever pushed onto the stack. The stack initially holds $|\mathcal{G}|$ vertices, and the algorithm can also push already-processed vertices back onto it (*re-adds*). These arise in two ways.

First, at the transition from type A to type B, which may occur at either of two sites in the canonicalization routine (Step 2 and Step 4b). Once the graph is type B it cannot become type A again, since there is no way to remove length-2 legs, so the two sites share a single budget of at most $|\mathcal{G}|$ re-adds, no vertex being re-added more than once.

Second, when the long leg of a type B graph shrinks, from length 4 to 3 or from 3 to 2. Here the re-added vertex is pushed onto the top of the stack and popped in the following iteration, where it is adjacent both to the end of the long leg and to the vertex attached to the core, and is therefore attached as the second vertex of a length-2 leg in $O(1)$ time. At most a constant number of such re-adds is produced per iteration, so each can be charged to the iteration that produced it.

The main loop therefore runs $O(|\mathcal{G}|)$ iterations of cost $O(|\mathcal{G}|)$ together with $O(|\mathcal{G}|)$ iterations of cost $O(1)$, giving $O(|\mathcal{G}|^2)$ basic operations.

Final dependency removal. After canonicalization, we drop

length-1 leg generators that are products of others. The bit-packed symplectic encoding gives a bijection between Pauli strings on n qubits and $\mathbb{F}_2$ vectors of length $2n$, so dependency removal is Gaussian elimination in $\mathbb{F}_2^{2n}$. We maintain the basis in reduced row-echelon form, indexed by pivot column so that the unique basis vector with a given pivot can be retrieved in $O(1)$. Reducing a new vector requires repeated XORs against the basis vector sharing its current pivot; since the pivot strictly advances with each XOR and lies in $\{1, \ldots, 2n\}$, reduction terminates in $O(n)$ XORs per vector. With $O(|\mathcal{G}|)$ length-1 legs to process, the dependency-removal step costs $O(n|\mathcal{G}|)$ basic operations.

Total. Summing the two phases:

Phase	**Basic operations**	**Word operations**
Main loop	$O(\|\mathcal{G}\|^2)$	$O(n\|\mathcal{G}\|^2)$
Final dependency removal	$O(n\|\mathcal{G}\|)$	$O(n^2\|\mathcal{G}\|)$
Total	$O(\|\mathcal{G}\| \max(n, \|\mathcal{G}\|))$	$O(n\|\mathcal{G}\| \max(n, \|\mathcal{G}\|))$

The word-operation column applies the cost model above: each basic operation (Pauli multiplication, commutation, or $\mathbb{F}_2^{2n}$ XOR) costs $O(n/W)$ word operations on the bit-packed symplectic encoding. In total,

$$T(n, |\mathcal{G}|) \;=\; \mathcal{O}(n|\mathcal{G}|(n + |\mathcal{G}|)) = O(n|\mathcal{G}| \max(n, |\mathcal{G}|)).$$

Connectivity. We may choose to initialize the stack with an arbitrary ordering of vertices, but this may lead to a situation where the vertex to be added is disconnected from the intermediate graph. In this situation, we use a queue instead of a stack, and thus we push the current vertex to the end of the queue. We do the same for re-adds. However, given that the cost of adding a new vertex is linear in the number of already added vertices, it is simple to show that no matter how many passes are required, since at least one vertex is added in each pass, and since the cost of skipping a vertex is $O(1)$, the total cost remains bounded by $O(n|\mathcal{G}|^2)$. In the worst case, a re-add requires us to rebuild almost the whole graph from scratch, but this only happens once, so the total cost continues to hold. Finally, it is possible to prove with some effort that initializing the stack with a DFS preorder ensures that connectivity is never lost. However, in light of the above fact, we have omitted this proof.

c) Example execution of algorithm: We classify the generator set of Figure 1 using the DFS preorder $S = [ZZII,\ YIIX,\ YXIZ,\ ZIII,\ YYXX,\ XIII,\ YIIZ,\ YYIX]$. Table II traces each iteration and Figure 3 illustrates the corresponding graph transformations. The final canonical graph has legs of lengths $\{1, 1, 2, 3\}$, so by Table I the algebra is $\mathfrak{su}(8) \oplus \mathfrak{su}(8)$ (type B3 with $n_c = 1$, $n_2 = 1$).

IV. Software Design and Architecture

PauLie's architecture decouples the memory representation of operators from their algebraic properties, so that the user-facing API tracks the underlying mathematical formalism while the bit-level backend can be optimized or ported independently. Figure 5 summarizes the principal classes; the description below walks through the two layers it depicts.

The operator and collection layer lives in `paulie.common`. `PauliString` stores an n-qubit operator as a single bit array of length $2n$ in its symplectic representation, backed by the dedicated `pauliebits` library, so that multiplication, commutation, and the adjoint action all reduce to bitwise XOR and parity computations—producing the $O(n/W)$ basic-operation cost analyzed in Section III. Figure 4 benchmarks these primitives against `Stim` [27] and `PauliArray` [28].

`PauliStringCollection` models a generator set and provides two entry points to the pipeline: `get_algebra` returns the DLA as a string label, while `classify` returns the full `Classification` object for callers that need the canonical graphs or the DLA dimension. Construction across both classes is mediated by an overloaded `get_pauli_string` factory.

The classification pipeline lives in `paulie.classifier`. The `build_canonical_graph` method of the `Canonicalizer` class consumes a DFS-ordered vertex stack from one connected component and produces a `Morph`—a typed canonical-graph object that records the legs and exposes the type label and leg counts needed to read off the algebra family from Table I. `Classification` aggregates the per-component `Morph`s, as the vertices of the canonical graph, into the direct-sum decomposition. Downstream functionality such as `get_optimal_universal_generators` lives in a separate `application` subpackage that consumes only the public interfaces of `PauliStringCollection` and `Classification`, so new applications can be added without modifying core classes and the bit-packed backend can be swapped (for example for a Cython or Rust implementation) without touching them.

V. Benchmarking

We validated the implementation by reproducing the tabulated results on DLAs generated by two-local Pauli strings [4] in `test_classification.py`. We then benchmarked PauLie against a brute-force Lie closure algorithm on random k-local Pauli Hamiltonians with $|\mathcal{G}| = (4 - k)n$ terms. The runtime comparison is shown in Figure 6. At $n = 20$ qubits PauLie completes the classification in $390\,\mu$s for $k = 2$ and $223\,\mu$s for $k = 3$, while the brute-force baseline already takes $2.4\,$ms at $n = 7$.

To characterise the empirical scaling of `get_algebra` we employ the *doubling-ratio* diagnostic, a non-parametric method for empirical complexity analysis [29, 30]. For any asymptotic scaling $T(n) \sim c\, n^a$, the ratio of runtimes at input sizes n and $2n$ satisfies

$$\log_2 \left[\frac{T(2n)}{T(n)} \right] \xrightarrow{n \to \infty} a, \tag{1}$$

so the effective exponent can be read directly off the y-axis as a function of n, without fitting a parametric model and without

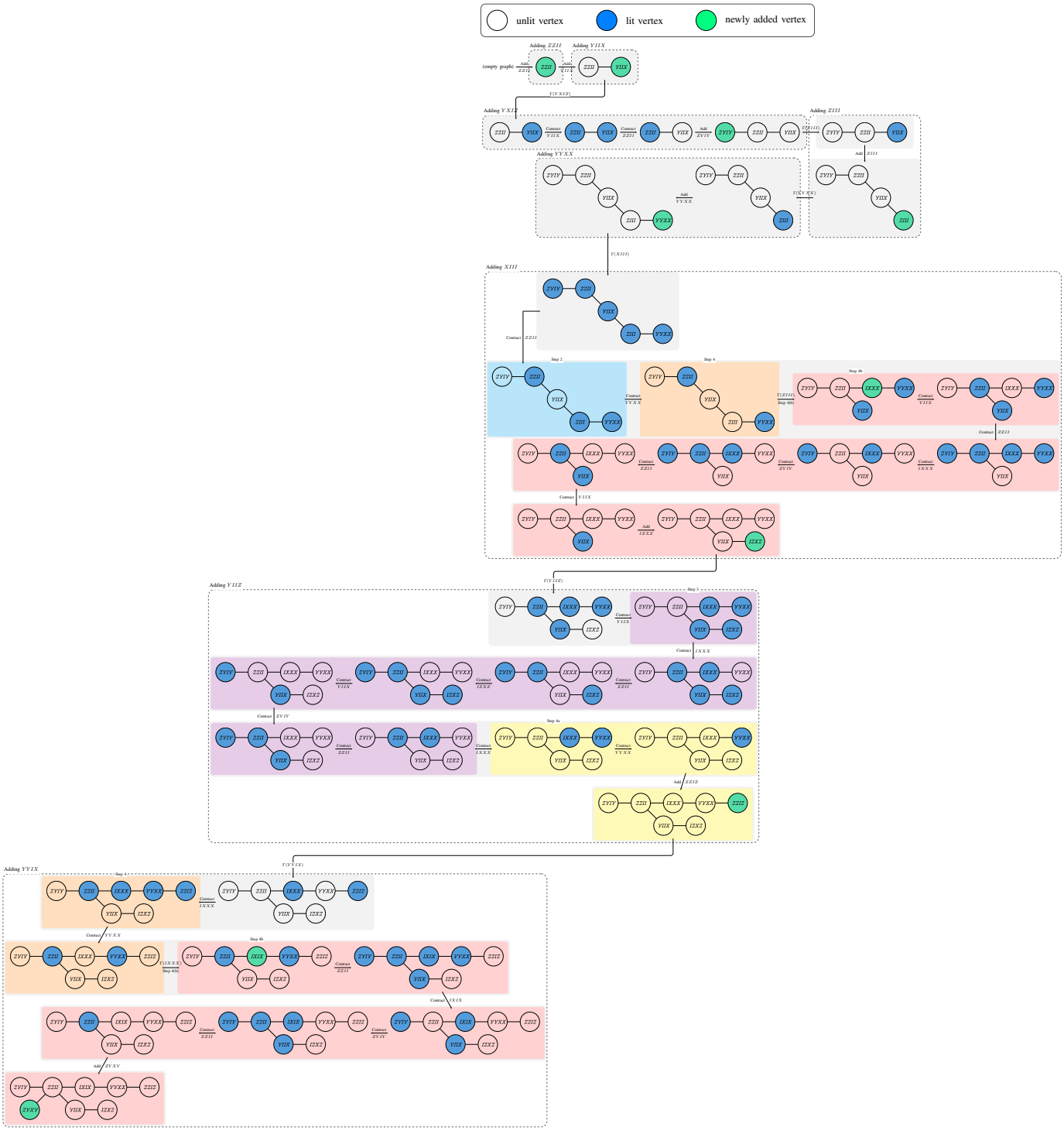

Figure 3. A visual walkthrough of the example in Table II.

Table II

TRACE OF ALGORITHM 1 ON THE GENERATORS OF FIGURE 1. EACH BLOCK OF ROWS CORRESPONDS TO ONE POPPED VERTEX; SUB-ROWS LIST CONTRACTIONS PERFORMED WITHIN THAT ITERATION. THE NOTATION $\cdot P$ DENOTES CONTRACTION WITH P, AND $\cdot(P_1, \ldots, P_k)$ DENOTES A LIGHTNING SEQUENCE APPLIED IN ORDER.

Popped V	Branch / sub-step	Operation	Effect
$ZZII$	Step 1: seed core	—	$C \leftarrow ZZII$
$YIIX$	Step 1: <2 legs, attach to core	—	first length-1 leg
$YXIZ$	Step 1: <2 legs, attach to core	$\cdot YIIX$, $\cdot ZZII$	$YXIZ \to ZYIY$; second length-1 leg
$ZIII$	extend $YIIX$ leg	—	longest leg length 2 (type A)
$YYXX$	extend longest leg	—	longest leg length 3 (type A)
$XIII$	Step 2: skipped (legs share lit state); normalize	$\cdot ZZII$	$XIII \to YZII$
	Step 3: skipped (no length-2 legs)	—	—
	Step 4: long leg has lit vertices	$\cdot YYXX$	$YZII \to IXXX$
	Step 4b: lightning relocates $ZIII$	$\cdot(YIIX, ZZII, IXXX, ZYIY, ZZII, YIIX)$	$ZIII \to IZXZ$; type B, two length-2 legs
$YIIZ$	Step 3: length-2 leg has lit vertex	$\cdot YIIX$	$YIIZ \to IIIY$
	Step 3 (cont.)	$\cdot IXXX$	$IIIY \to IXXZ$, central lit
	Step 3 (cont.)	$\cdot(ZZII, IZXZ, YIIX, ZYIY, ZZII)$	$IXXZ \to XIIZ$
	Step 4a: first vertex of long leg lit	$\cdot IXXX$, $\cdot YYXX$	$XIIZ \to ZZIZ$, attached to $YYXX$
$YYIX$	Step 4: light central vertex	$\cdot IXXX$	$YYIX \to YZXI$
	Step 4 (cont.): second-last of long leg lit	$\cdot YYXX$	$YZXI \to IXIX$
	Step 4b: lightning relocates $IXXX$	$\cdot(ZZII, IXIX, ZYIY, ZZII)$	$IXXX \to ZYXY$; legs $\{1, 1, 2, 3\}$, type B3

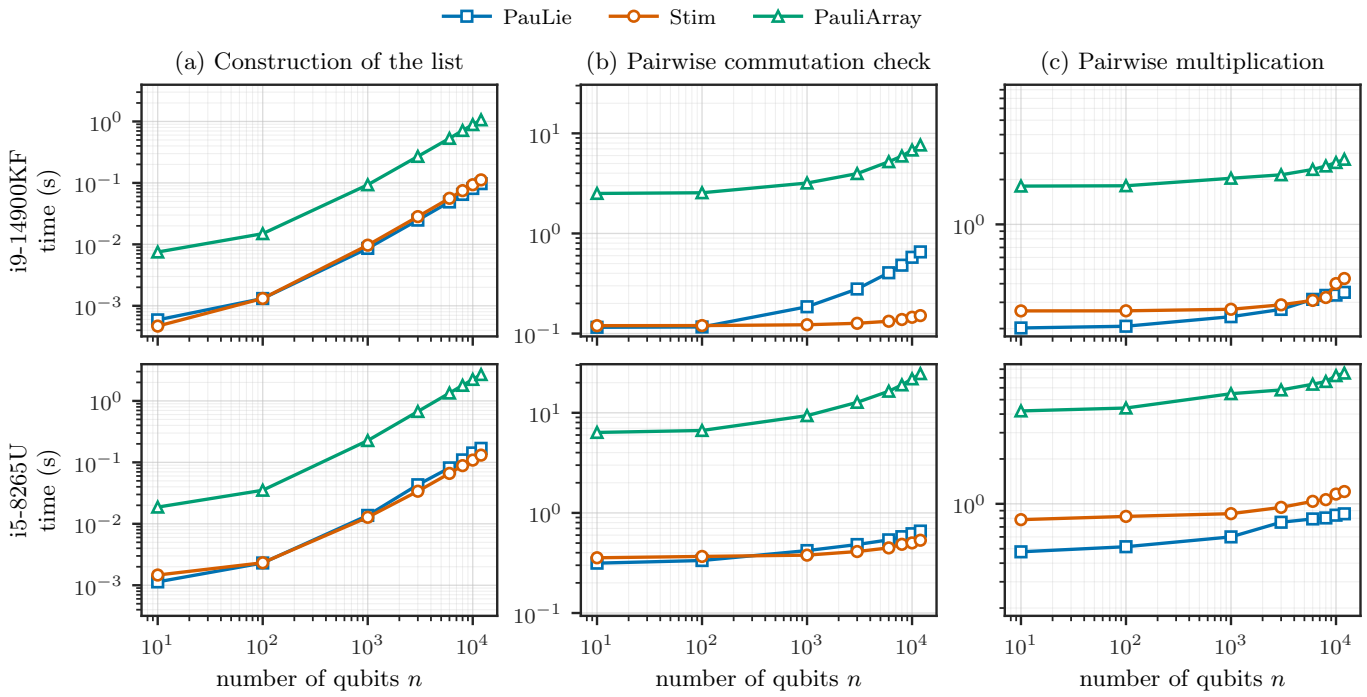

Figure 4. Backend primitives as a function of the number of qubits, on two processors (rows), for a list of 1000 random Pauli strings: construction of the list (a), and all $\binom{1000}{2}$ pairwise commutation checks (b) and pairwise multiplications (c). Each point is the minimum over 100 repetitions on the i9-14900KF and over 30 on the i5-8265U. Both axes are logarithmic and the two rows share a vertical scale within each column. PauLie leads on multiplication at all but two measured points and is level with `Stim` on construction, the small residual lead flipping between the two processors; on commutation it is overtaken by `Stim` for $n \gtrsim 10^3$, where `Stim`'s cost on the i9-14900KF is nearly flat in n.

small-n overhead affecting the large-n estimate. Figure 7 shows the measured ratios for random k-local Pauli Hamiltonians. Across the measured range, the ratio rises monotonically and approaches 2 at the largest accessible n, remaining well

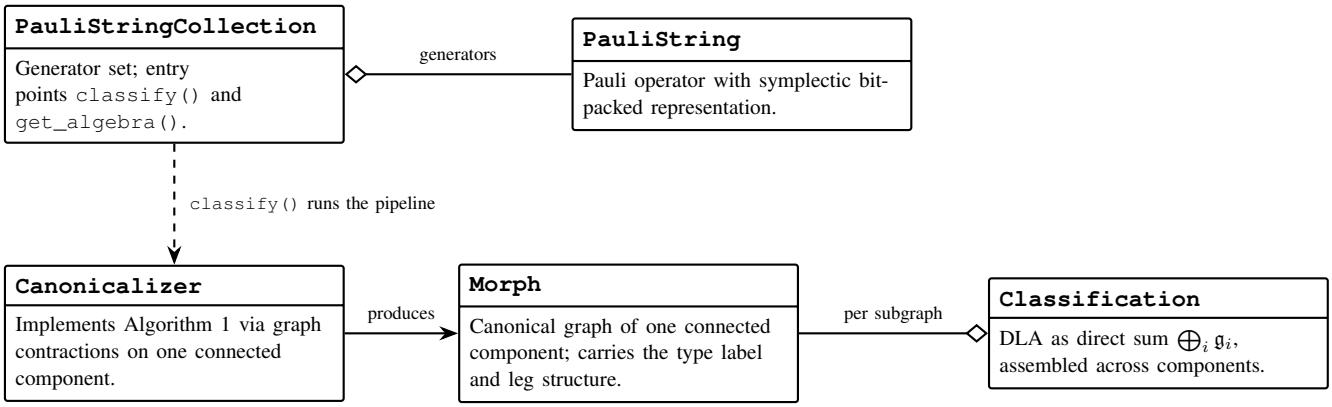

Figure 5. Class diagram of the PauLie classification pipeline. A `PauliStringCollection` holds a generator set of `PauliString`s. Calling `classify()` on the collection instantiates a `Canonicalizer` and a `Classification`; for each connected component of the anticommutation graph, the `Canonicalizer` (Algorithm 1) builds a `Morph` that is added to the `Classification`. The `Classification` is stored as an attribute of the `PauliStringCollection` and assembles the per-component `Morph`s, as the vertices of the canonical graph, into the direct-sum decomposition of the DLA. A diamond at one end of a connection marks the container side of a "contains-many" relationship (a `PauliStringCollection` contains many `PauliString`s; a `Classification` contains many `Morph`s). The solid arrow marks the `Canonicalizer`'s `Morph` output, and the dashed arrow marks the runtime call that drives the pipeline.

below the $a = 3$ reference throughout. This is consistent with the theoretical worst-case bound of $O(n|\mathcal{G}|\max(n, |\mathcal{G}|))$ and demonstrates that the typical-case runtime on random k-local Hamiltonians is substantially more favourable.

VI. Applications of DLA Classification

The classification is fast enough to be consulted inside larger workflows; this section collects such applications.

A. Lie-Algebraic Simulation

Synthesizing unitaries via exact Cartan decomposition requires a Lie algebra with a compact representation [9]. Likewise, classical simulation of quantum systems is efficient in the polynomial-sized DLA regime [10]. Classifying the generator set first ensures that these algorithms are invoked only for polynomial-sized DLAs, while exponentially large algebras are handed to approximate simulation techniques.

B. Generator-Set Engineering

PauLie also supports the engineering of generator sets. Concretely, we identify generator sets consisting of Pauli strings with optimal generation rate for any dimension. These are optimal in the sense that the number of Pauli strings generated in successive rounds of nested commutators grows maximally. As shown in [31], optimal generating sets are characterized by having a fraction of anticommuting generator pairs close to 0.706. Moreover, the minimal number of Pauli-string generators required to generate $\mathfrak{su}(2^n)$ is $2n + 1$. This allows the problem to be formulated in terms of the anticommutation graph. Optimal sets therefore correspond to graphs with approximately

$$\left\lfloor 0.706 \cdot \binom{2n+1}{2} \right\rfloor$$

edges.

The search starts from the minimal universal set of [31, Example 1]. The seed's anticommutation fraction lies far below the optimal 0.706. Repeatedly replacing a generator P by its product PQ with an anticommuting neighbour Q preserves minimality and universality while rewiring the anticommutation graph, and replacements are applied until the number of anticommuting pairs reaches the target above. The search is implemented in `get_optimal_universal_generators`. Beyond such concrete questions on gate sets, PauLie can be employed to study theoretical algebraic questions. Allcock et al. [15] demonstrated how to construct direct sums of multiple copies of a DLA with only logarithmically many extra qubits. PauLie identifies the required anticommutation structures directly, so these direct-power constructions can be verified, and related open questions probed, numerically.

C. Trainability of variational quantum algorithms

In quantum machine learning, the DLA controls the onset of barren plateaus [13, 14]. Kazi et al. [12] use Pauli DLA classifications to analyze multi-angle MaxCut-QAOA on connected graphs. For $G = (V, E)$, the generator set is

$$\mathcal{G}_{\text{free}} = \{X_v : v \in V\} \cup \{Z_u Z_v : \{u, v\} \in E\},$$

and $\mathfrak{g}_{\text{free}} = \langle i\mathcal{G}_{\text{free}} \rangle_{\text{Lie}}$.

For this multi-angle ansatz, [12, Theorem 1] classifies the DLA for every connected graph with $n \geq 2$ vertices. The six cases are: paths; cycles; connected bipartite graphs that are neither paths nor cycles, split according to whether the bipartition sizes are even-even, even-odd, or odd-odd; and archetypal graphs, i.e., connected graphs that are neither bipartite nor cycles. For representatives of each of the six graph families at increasing n, PauLie returns the predicted algebra and dimension, as shown in Figure 8.

[12, Corollary 2] connects the DLA classification to trainability for archetypal graphs with $n > 3$. If a sufficiently deep

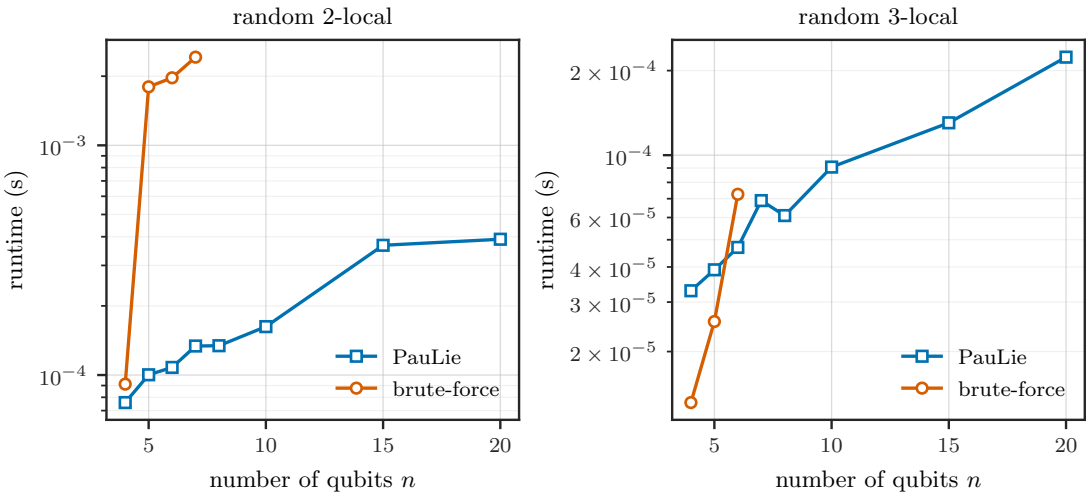

Figure 6. Runtime of PauLie's classification and of brute-force closure on random k-local generators. The baseline returns a basis and PauLie returns an isomorphism type. Each point is the minimum over 7 repeated timings. The brute-force baseline is capped at 30 s per instance.

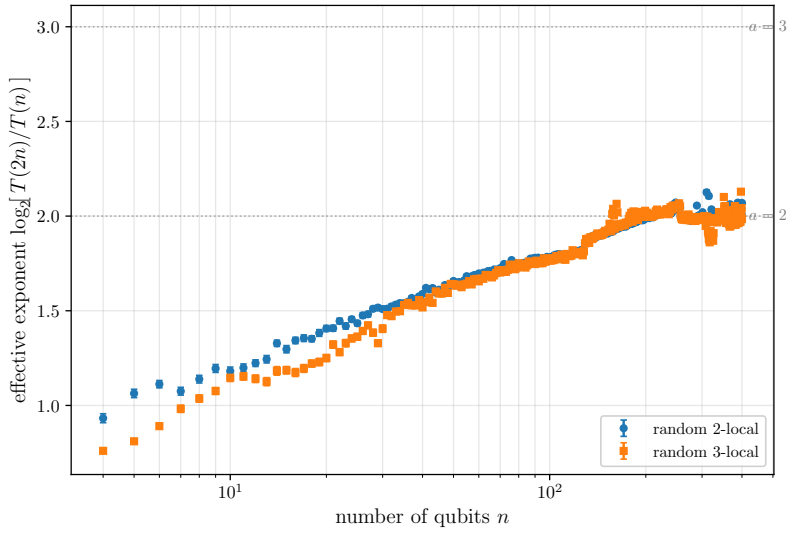

Figure 7. **Empirical effective exponent of `get_algebra` via the doubling-ratio method.** For each n such that both $T(n)$ and $T(2n)$ have been measured, the quantity $\log_2[T(2n)/T(n)]$ converges to the exponent a of any asymptotic scaling $T(n) \sim c\,n^a$. Runtimes are measured for random k-local Pauli Hamiltonians with $k \in \{2, 3\}$ and $n \in \{4, 5, \ldots, 800\}$, using 100 independent random instances per n; each instance is denoised by taking the minimum over 3 repeated timings of `get_algebra`. Error bars show propagated standard error of the mean of $T(n)$ and $T(2n)$. Dotted grey lines mark the values $a = 2$ and $a = 3$ for reference.

multi-angle QAOA circuit induces an approximate unitary 2-design, then, with $d = 2^n$,

$$\mathrm{Var}_{\boldsymbol{\theta}}[\partial_\vartheta C(\boldsymbol{\theta})] = \frac{4d^2|E|}{(d^2-4)(d+2)} \leq \frac{4n^2}{2^n}. \tag{2}$$

Gradients thus vanish exponentially in n: the barren-plateau phenomenon.

The comparison is confined to the multi-angle variant, whose generators are individual Pauli strings. With shared angles each generator becomes a linear combination of Pauli strings; the generated algebra is then contained in the one generated by the individual terms, but in general strictly smaller and coefficient-dependent, so PauLie's classification does not apply directly. Mao et al. [32] treat that regime for MaxCut and find a DLA dimension of $\Theta(4^n)$ for almost all graphs.

VII. Discussion and Future Work

PauLie brings DLA classification into the range of problem sizes relevant to current quantum algorithm research. Systems that previously required case-by-case analysis can now be classified routinely, which we expect to change how DLA structure enters algorithm design.

The framework is well suited to act as a routing oracle upstream of heavier methods. Exact Cartan decomposition

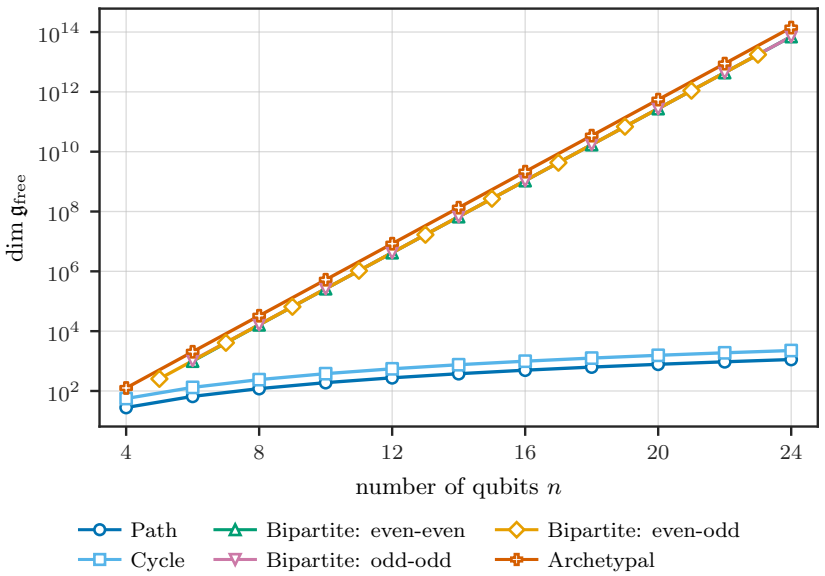

Figure 8. Analytic multi-angle MaxCut-QAOA DLA dimensions from Kazi et al. (solid curves) and PauLie classifications (open markers) for representative graphs from the path, cycle, bipartite parity, and archetypal classes.

[9], classical simulation of polynomial-DLA systems [10], and barren-plateau analysis of variational ansätze [12] all depend on the DLA. PauLie determines the DLA efficiently, ensuring that approximate methods are applied only when necessary. The same capability supports numerical investigation of questions previously accessible only through manual analysis, such as generator-set optimality [31] and the construction of generator sets whose DLA is a direct power $\mathfrak{g}^{\oplus k}$ of a given algebra, using only logarithmically many extra qubits [15].

Several extensions follow naturally. Since the canonical graphs obtained by PauLie are perfect, they are in particular $\hbar$-perfect in the sense of [33]. Consequently, the full set of applications developed there applies directly to the canonical graphs: efficient entanglement-detection schemes, connections to the complexity of shadow tomography, tight uncertainty relations, and constructions of strong lower bounds on ground-state energies. In particular, the Pauli strings labelling the vertices of a canonical graph span the operator subspace in which corresponding entanglement-witness operators can be constructed. Perfection also certifies tractability in magic detection. For Pauli measurement sets whose anticommutation graph is perfect and free of active sign dependencies, Liu et al. [34] show that the reduced stabilizer polytope becomes efficiently solvable, with witness capacity bounded by the clique number. The canonical generator sets are thus natural candidates for scalable nonstabilizerness witnesses. For all of these applications, an important open question is to what extent they can be pulled back from the canonical representative to the original instance.

Further extensions concern the class of supported inputs. Termwise Hamiltonian encodings in the sense of [35] preserve the DLA isomorphism. Since the encoded image of a Pauli string may have a nontrivial Pauli expansion, this provides a principled route towards DLA classifications for generators that are linear combinations of Pauli strings. Beyond qubits, the graph-theoretic backend may also be extended to generalized Pauli observables through the qudit frustration-graph formalism of [16].

Lastly, the implementation itself can be extended. PauLie returns the isomorphism type of the DLA rather than an explicit basis, so applications that consume a basis still require closure. Producing bases in the cases where the algebra is small enough for one to be useful is a natural next step. Also, building the anticommutation graph requires $\binom{|\mathcal{G}|}{2}$ pairwise commutation checks, each a parity computation on the bit-packed representation, and all of them are independent of one another. Li et al. [36] implement precisely this kind of bitwise Pauli arithmetic on GPUs. Their framework could therefore take the place of PauLie's backend and evaluate the checks in parallel.

Acknowledgments

The authors thank the Unitary Foundation for supporting the package through two microgrants and mentorship. We are grateful to Hendrik Poulsen Nautrup for suggesting the framework's application to discovering optimal universal generator sets. We further acknowledge Tarin Teacharsripaitoon for assistance with code quality improvements, as well as the open-source community for their contributions.

References

[1] Robert Zeier and Thomas Schulte-Herbrüggen. "Symmetry-directed control of quantum dynamical systems". In: *Physical Review A* 83.4 (Apr. 2011), p. 042314. DOI: 10.1103/PhysRevA.83.042314. URL: https://doi.org/10.1103/PhysRevA.83.042314.

[2] Domenico D'Alessandro. *Introduction to Quantum Control and Dynamics*. CRC Press, 2007. ISBN: 9781584888833.

[3] Velimir Jurdjevic and Héctor J. Sussmann. "Control systems on Lie groups". In: *Journal of Differential Equations* 12.2 (1972), pp. 313–329. DOI: 10.1016/0022-0396(72)90035-6.

[4] Roeland Wiersema, Efekan Kökcü, Alexander F. Kemper, et al. "Classification of dynamical Lie algebras of 2-local spin systems on linear, circular and fully connected topologies". In: *npj Quantum Information* 10 (2024), p. 110. DOI: 10.1038/s41534-024-00900-2. URL: https://doi.org/10.1038/s41534-024-00900-2.

[5] Efekan Kökcü et al. *Classification of dynamical Lie algebras generated by spin interactions on undirected graphs*. 2024. DOI: 10.48550/arXiv.2409.19797. arXiv: 2409.19797 [quant-ph]. URL: https://arxiv.org/abs/2409.19797.

[6] Gerard Aguilar et al. *Full classification of Pauli Lie algebras*. 2024. DOI: 10.48550/arXiv.2408.00081. arXiv: 2408.00081 [quant-ph]. URL: https://arxiv.org/abs/2408.00081.

[7] Oxana Shaya et al. *PauLie*. https://github.com/QPauLie/PauLie. 2026.

[8] Hans Cuypers. *Dynamical Lie algebras generated by Pauli strings and quadratic spaces over* $\mathbb{F}_2$. 2026. DOI: 10.48550/arXiv.2603.08373. arXiv: 2603.08373 [quant-ph]. URL: https://arxiv.org/abs/2603.08373.

[9] David Wierichs et al. *Recursive Cartan decompositions for unitary synthesis*. 2025. arXiv: 2503.19014 [quant-ph]. URL: https://arxiv.org/abs/2503.19014.

[10] Matthew L. Goh et al. *Lie-algebraic classical simulations for quantum computing*. 2023. DOI: 10.48550/arXiv.2308.01432. arXiv: 2308.01432 [quant-ph]. URL: https://arxiv.org/abs/2308.01432.

[11] M. Cerezo et al. *Does provable absence of barren plateaus imply classical simulability? Or, why we need to rethink variational quantum computing*. 2023. DOI: 10.48550/arXiv.2312.09121. arXiv: 2312.09121 [quant-ph]. URL: https://arxiv.org/abs/2312.09121.

[12] Sujay Kazi et al. *Analyzing the quantum approximate optimization algorithm: ansätze, symmetries, and Lie algebras*. 2024. DOI: 10.48550/arXiv.2410.05187. arXiv: 2410.05187 [quant-ph]. URL: https://arxiv.org/abs/2410.05187.

[13] Michael Ragone et al. *A Lie Algebraic Theory of Barren Plateaus for Deep Parameterized Quantum Circuits*. 2023. DOI: 10.48550/arXiv.2309.09342. arXiv: 2309.09342 [quant-ph]. URL: https://arxiv.org/abs/2309.09342.

[14] Enrico Fontana et al. *The Adjoint Is All You Need: Characterizing Barren Plateaus in Quantum Ansätze*. 2023. DOI: 10.48550/arXiv.2309.07902. arXiv: 2309.07902 [quant-ph]. URL: https://arxiv.org/abs/2309.07902.

[15] Jonathan Allcock et al. *On generating direct powers of dynamical Lie algebras*. 2025. DOI: 10.48550/arXiv.2506.05733. arXiv: 2506.05733 [quant-ph]. URL: https://arxiv.org/abs/2506.05733.

[16] Owidiusz Makuta, Błażej Kuzaka, and Remigiusz Augusiak. *Frustration graph formalism for qudit observables*. 2025. DOI: 10.48550/arXiv.2503.22400. arXiv: 2503.22400 [quant-ph]. URL: https://arxiv.org/abs/2503.22400.

[17] Maxwell West et al. *A graph-theoretic approach to chaos and complexity in quantum systems*. 2025. DOI: 10.48550/arXiv.2502.16404. arXiv: 2502.16404 [quant-ph]. URL: https://arxiv.org/abs/2502.16404.

[18] Seth Lloyd. "Universal Quantum Simulators". In: *Science* 273 (1996), pp. 1073–1078. DOI: 10.1126/science.273.5278.1073.

[19] Rolando Somma et al. "Simulating physical phenomena by quantum networks". In: *Physical Review A* 65 (2002), p. 042323. DOI: 10.1103/PhysRevA.65.042323.

[20] Domenico D'Alessandro. *Introduction to Quantum Control and Dynamics*. 2nd ed. Chapman and Hall/CRC, 2021. DOI: 10.1201/9781003051268.

[21] Francesca Albertini and Domenico D'Alessandro. "The Lie algebra structure and controllability of spin systems". In: *Linear Algebra and its Applications* 350 (2002), pp. 213–235. DOI: 10.1016/S0024-3795(02)00290-2.

[22] Dave Wecker, Matthew B. Hastings, and Matthias Troyer. "Progress towards practical quantum variational algorithms". In: *Physical Review A* 92 (2015), p. 042303. DOI: 10.1103/PhysRevA.92.042303.

[23] Martin Larocca et al. "Diagnosing Barren Plateaus with Tools from Quantum Optimal Control". In: *Quantum* 6 (2022), p. 824. DOI: 10.22331/q-2022-09-29-824.

[24] M. Cerezo et al. "Variational Quantum Algorithms". In: *Nature Reviews Physics* 3 (2021), pp. 625–644. DOI: 10.1038/s42254-021-00348-9.

[25] Velimir Jurdjevic and Héctor J. Sussmann. "Control Systems on Lie Groups". In: *Journal of Differential Equations* 12.2 (1972), pp. 313–329. DOI: 10.1016/0022-0396(72)90035-6.

[26] Yutaro Iiyama. *Fast numerical generation of Lie closure*. 2025. DOI: 10.48550/arXiv.2506.01120. arXiv: 2506.01120 [quant-ph]. URL: https://arxiv.org/abs/2506.01120.

[27] Craig Gidney. "Stim: a fast stabilizer circuit simulator". In: *Quantum* 5 (July 2021), p. 497. DOI: 10.22331/q-2021-07-06-497. URL: https://doi.org/10.22331/q-2021-07-06-497.

[28] Maxime Dion, Tania Belabbas, and Nolan Bastien. *Efficiently manipulating Pauli strings with PauliArray*. 2024. arXiv: 2405.19287 [quant-ph]. URL: https://arxiv.org/abs/2405.19287.

[29] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley Professional, 2011.

[30] Catherine McGeoch and Gregory M. Kapfhammer. "EXPOSE: Inferring Worst-case Time Complexity by Automatic Empirical Study". In: *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 2015.

[31] Isaac D. Smith et al. *Optimally generating $\mathfrak{su}(2^N)$ using Pauli strings*. 2024. DOI: 10.48550/arXiv.2408.03294. arXiv: 2408.03294 [quant-ph]. URL: https://arxiv.org/abs/2408.03294.

[32] Rui Mao et al. *QAOA-MaxCut has barren plateaus for almost all graphs*. 2025. arXiv: 2512.24577 [quant-ph]. URL: https://arxiv.org/abs/2512.24577.

[33] Zhen-Peng Xu et al. *Simultaneous variances of Pauli strings, weighted independence numbers, and a new kind of perfection of graphs*. 2025. arXiv: 2511.13531 [quant-ph]. URL: https://arxiv.org/abs/2511.13531.

[34] Yingjian Liu et al. *Graph Theoretic Approach to Quantum Nonstabilizerness*. 2026. arXiv: 2607.26154 [quant-ph]. URL: https://arxiv.org/abs/2607.26154.

[35] Toby S. Cubitt, Ashley Montanaro, and Stephen Piddock. "Universal quantum Hamiltonians". In: *Proceedings of the National Academy of Sciences* 115.38 (2018), pp. 9497–9502. DOI: 10.1073/pnas.1804949115. URL: https://doi.org/10.1073/pnas.1804949115.

[36] Xiaopeng Li et al. *A High-Performance Pauli-Algebra Framework for Large-Scale Quantum Simulations*. 2026. arXiv: 2606.28952. URL: https://arxiv.org/abs/2606.28952.