Data structures for quantum circuit verification and how to compare them

Vinkhuijzen, L.T.

Citation

Vinkhuijzen, L. T. (2025, February 25). *Data structures for quantum circuit verification and how to compare them*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/4208911

Version: Publisher's Version

License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden

Downloaded from: https://hdl.handle.net/1887/4208911

Note: To cite this publication please use the final published version (if applicable).

Data Structures for Quantum Circuit Verification and How To Compare Them

Lieuwe Thomas Vinkhuijzen

Data Structures for Quantum Circuit Verification and How To Compare Them

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 25 februari 2025
klokke 14.30 uur

door
Lieuwe Thomas Vinkhuijzen

geboren te Alphen aan den Rijn

in 1993

Promotor: Prof.dr. H.H. Hoos

Co-promotores: Dr. A.W. Laarman
Dr. T.J. Coopmans

Promotiecomissie:

Dr. J. Tura Brugués	
Prof.dr. H.C.M. Kleijn	
Prof.dr. A. Plaat	
Prof.dr. W.J. Fokkink	(Vrije Universiteit Amsterdam)
Dr. K.S. Meel	(University of Toronto)
Prof.dr. R. Wille	(Technische Universität München)

Copyright © 2025 Lieuwe Vinkhuijzen.

IPA Dissertation Series 2025-05

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

This project was funded by the research program VENI with project number 639.021.649 of the Netherlands Organization for Scientific Research (NWO).

Cover art: *Colourful Linings*, Gouache on canvas, by Joyce Olivier

Acknowledgements

First, I wish to thank my copromotor Alfons Laarman for his supervision. I am grateful for the wide creative latitude you gave me in pursuing directions and ideas in my research. You always took the effort to understand these ideas at least as well as I did. As a result, your technical feedback was always superb, abundant and on point. I am indebted to Tim Coopmans for being an excellent copromotor in the final days of my PhD project. Our long collaboration on LIMDDs has been an extremely enjoyable part of my time here. I thank my promotor Holger Hoos for his mentorship, for many insightful discussions and many useful comments on drafts of all my papers.

I am greatly indebted to all the folks at the Johannes Kepler Universität in Linz, Austria. Robert Wille, Stefan Hillmich and Thomas Grurl, thank you for your hospitality and for being fantastic coauthors. I thank Richard Kueng for insightful discussions; Tom Peham for explaining about dolphins; and Aaron Sander and Lukas Burgholzer for the most opinionated discussions anyone will ever have about decision diagrams. The helpful and friendly staff at the JKU student accommodations made my stay especially pleasant.

I thank my coauthors Vedran Dunjko and David Elkouss for your involvement in our LIMDD paper, and for your patience during the long design phase.

I thank André Deutz for his invaluable support.

My time at LIACS would not have been the same without many other PhD students who were in the same boat and who provided lunch + fun; in no particular order, Marie, Marios, Koen, Arina, Tanjona, Anna, Anne, Lise, Sander, Thodoris, Hugo, Can, Dimitrios, Sebastiaan, António, Alexa, Leni, Daniela, Charles, Solomiia, Grace, Marshall, Yash, Furong, Thomas, Gerrit Jan, Casper, Rachel, Dalia, Niki, Hao, thank you for all the lunches, the music and the pastries.

Many anonymous reviewers have made insightful and valuable comments on drafts of our papers. I am indebted for their help both in rigorously checking the mathematical and technical details and in improving the presentation of our work, even when our paper wasn't accepted for publication. Therefore, I wish to thank the anonymous reviewers at QIP, SODA, STOC and Quantum for their feedback on drafts of Chapter 3. I thank the anonymous reviewers at ASP-DAC and SPIN for their feedback on drafts of Chapter 4. I thank the anonymous reviewers at KR, IJCAR, QIP and AAAI for their helpful feedback on drafts of Chapter 5. I thank the anonymous NFM reviewers for helpful feedback on a draft of Chapter 6. I thank the anonymous reviewers at CONCUR and SETTA for valuable feedback on drafts of Chapter 7. I moreover thank the organizers of SETTA for sending a substantial amount of Chinese tea halfway around the world by way of a prize for the best video presentation award – I am still enjoying it.

Lastly, I want to thank all the functions, polynomials, Boolean circuits, quantum circuits, algebraic circuits, all the directed acyclic weighted graphs, and the estimates, inequalities and eigenvalues, without which this thesis would be all the poorer.

Contents

Acknowledgements **v**

1 Introduction **1**

1.1 Context 2

1.2 State-of-the-art data structures for quantum algorithm analysis 6

1.3 Research questions 12

1.4 Contributions 13

1.5 Outline 15

1.6 Publications and artefacts 16

2 Preliminaries **19**

2.1 Mathematical preliminaries 20

2.2 Introduction to quantum computing 21

2.2.1 Quantum states and operations 24

2.2.2 How to simulate a quantum circuit 27

2.2.3 Clifford circuits, stabilizer states and Pauli operators 27

2.3 Decision Diagrams 30

2.3.1 Decision diagrams for quantum states 33

2.3.2 Comparing decision diagrams 37

2.4 Approaches to quantum software verification 39

3 LIMDD: a decision diagram for simulation of quantum computing including stabilizer states 45

3.1 Introduction 46

3.2 Preliminaries: decision diagrams and the stabilizer formalism 49

3.2.1 Decision diagrams 49

3.2.2 Pauli operators and stabilizer states 52

3.2.3 Matrix product states 55

3.3 Local Invertible Map Decision Diagrams 55

3.3.1 The LIMDD data structure 57

3.3.2 Succinctness of LIMDDs 62

3.3.3 Pauli-LIMDD manipulation algorithms for simulation of quantum computing 67

3.3.4 Comparing LIMDD-based simulation with other methods . . . 79

3.4 Canonicity: Reduced LIMDDs with efficient MakeEdge algorithm . . 82

3.4.1 LIMDD canonical form 82

3.4.2 The MakeEdge subroutine: Maintaining canonicity during simulation 90

3.5 Efficient algorithms for choosing a canonical high label 93

3.5.1 Choosing a canonical high label 94

3.5.2 Efficient linear-algebra algorithms for stabilizer subgroups . . 97

3.5.3 Constructing the stabilizer subgroup of a LIMDD node 100

3.5.4 Efficiently finding a minimal LIM by multiplying with stabilizers 105

3.6 Numerical search for the stabilizer rank of Dicke states 110

3.7 Related work 111

3.8 Discussion 113

4 Efficient implementation of LIMDDs for Quantum Circuit Simulation 115

4.1 Introduction 116

4.2 Background 119

4.2.1 Quantum states and operations 119

4.2.2 Classical quantum circuit simulation 121

4.2.3 Verification of quantum circuits 123

4.3 Motivation 123

4.3.1 Quantum Multiple-valued Decision Diagrams (QMDDs) 124

4.3.2 Local Invertible Map Decision Diagrams (LIMDDs) 126

4.3.3 The need for a LIMDD implementation 127

4.4 Implementation of LIMDDs 128

4.4.1 Established techniques 129

4.4.2 Implementing Local Invertible Maps 130

4.5 Case study 133

4.6 Discussion 135

4.7 Conclusions 136

5 A Quantum Knowledge Compilation Map 137

5.1 Introduction 138

5.2 Data structures for quantum states 140

5.2.1 Quantum information 140

5.2.2 Data structures 141

5.3 Succinctness of quantum representations 145
5.4 Tractability of quantum operations 146
5.5 Rapidity of data structures 148
    5.5.1 A sufficient condition for Rapidity 149
    5.5.2 Rapidity between quantum representations 151
5.6 Related work 152
5.7 Discussion 154

6 The Power of Disjoint Support Decompositions in BDDs 157
6.1 Introduction 158
6.2 Background and related work 159
6.3 Succinctness separation between DSDBDD and BDD 163
6.4 Conclusion and future work 165

7 Model Checking with Sentential Decision Diagrams 167
7.1 Introduction 168
7.2 Background 170
    7.2.1 Sentential Decision Diagrams 170
    7.2.2 Model checking and reachability 173
7.3 Vtree heuristics 174
    7.3.1 Two-phase vtree construction 174
    7.3.2 Abstract variable tree heuristic 175
    7.3.3 Folding out vtrees for large variable domains 176
7.4 Related work 177
7.5 Experiments 177
    7.5.1 Results 179

7.5.2 SDD runtime profiles 182
7.6 Conclusion and future work 183

8 Conclusions and outlook 187

8.1 Research questions, revisited 187
8.2 Discussion 191
8.3 Future work 193
8.3.1 Future work on LIMDDs. 193
8.3.2 Future work on DSDBDDs 197
8.3.3 Future work on SDDs 198
8.3.4 Future work on quantum knowledge compilation 199
8.3.5 Future work on verification and decision diagrams 200

A Proof that cluster states and coset states need exponentially large QMDDs 205

B How to write graph states, coset states and stabilizer states as Tower-LIMDDs 211

C Advanced LIMDD algorithms 221

C.1 Measuring an arbitrary qubit 221
C.2 Applying Hadamards to stabilizer states in polynomial time 223

D LIMDDs prepare the W state efficiently 231

E Proofs of Section 5.3 239

F Proofs of Section 5.4 245

F.1 Easy and hard operations for ADD 245

F.2 Easy and hard operations for QMDD 247
F.3 Easy and hard operations for LIMDD 251
F.4 Easy operations for MPS 258
F.5 Easy and hard operations for RBM 259

G Proofs of Section 5.5 **261**

G.1 Rapidity is a preorder and generalizes earlier definitions 261
G.2 A Sufficient Condition for Rapidity 265
G.3 Rapidity Relations between Data Structures 267
G.4 MPS is at least as Rapid as QMDD 268
G.5 Transformations between QDDs 271
    G.5.1 Transforming LIMDD to QMDD 272
    G.5.2 Transforming QMDD to LIMDD 272
    G.5.3 Transforming ADD to QMDD 272
    G.5.4 Transforming QMDD to ADD 273

Bibliography **275**

Summary **305**

Samenvatting **309**

Curriculum Vitae **313**

Chapter 1

Introduction

Our world contains more and more software. Moreover, we are putting this software in charge of ever more important tasks in our daily lives. Software now operates or helps to operate traffic lights [277], medical devices [200], airplanes [171], spacecraft [151,160, 202] and innumerable more devices. A bug in this software can, in an innocent case, cause longer queues at the traffic light, and in the worst case can fail catastrophically when lives depend on it.

Quantum computing is a new and emerging field which may in the near or medium term deliver devices which can solve problems that are believed to be intractable on our current computers. To make quantum computing a reality, we need to be able to compile, optimize, design, synthesize, simulate and verify the correctness of quantum algorithms. Correctness verification in particular is a crucial part of the compilation and optimization of quantum algorithms (e.g., [13, 299]). However, verifying that software is correct is a tremendously difficult task, and becomes even more challenging if we wish to analyze software written for quantum computers instead of conventional ones. We therefore need powerful tools which can verify whether a proposed piece of software works as intended. In many of these approaches, the key ingredient is a data structure which can compactly store a quantum state. Therefore, this thesis has the following goal:

In this thesis, we introduce and analyze data structures for simulation and verification of software for quantum computers.

1.1 Context

Quantum computing. Physics tells us that nature is not classical, but *quantum* [139, 240]. This means that subatomic particles such as photons and electrons behave in ways that are different from the behaviour of large, everyday objects such as desks and coffee mugs. Specifically, particles can, in some sense, be in multiple places at once (called "superposition"), and they can interact and interfere with one another in ways that everyday objects do not (called "entanglement"). Current computers do not make use of these properties of matter, but scientists have speculated since the 1980s these phenomena may be useful for performing certain computations [119, 120]. A device which uses superposition and entanglement to its advantage would be called a *quantum computer*.

Quantum computing is a new and drastically different computing paradigm promising to efficiently solve several important problems that are believed to be intractable for classical computers. Examples of such problems include[*] integer factorization and discrete logarithm using Shor's algorithm [291], solving Pell's equation [149], computing the ideal class group and solving the principal ideal problem for a number field [50, 148], unstructured search using Grover's algorithm [12, 140, 231] and problems in quantum chemistry [172,196,201,208,219,274,314,340,364]. In the near term, so-called *Noisy Intermediate-Scale Quantum* (NISQ) devices are expected to both solve practical problems as well as deliver empirical evidence of quantum advantage over classical computers (see [267] for a recent perspective and [48] for a survey of NISQ devices and algorithms).

In order to achieve these lofty goals, it is not enough to be able to build large quantum computers. We will also need tools to, among other things, verify that the algorithms we design are correct; and to compile these quantum algorithms into correct implementations on hardware. These techniques are necessary to make quantum computing a practical reality; for example, compilation is necessary because the hardware typically imposes constraints on the connectivity of the qubits (e.g., [49, 287, 299, 371, 375]), or

---

[*] The Quantum Algorithm Zoo maintains an up-to-date list [175].

may support only a limited set of native quantum gates [75]; and circuit optimization is necessary to reduce the size and depth of a circuit, so that a computation can be performed using the limited resources of a small quantum computer [14, 286]. We speak of *analyzing quantum algorithms* as an umbrella term which includes the compilation, synthesis, optimization, simulation and verification, including equivalence checking and model checking, of quantum circuits. These tasks are interrelated; for example, some approaches to circuit optimization employ circuit equivalence checking to check whether a given optimization preserved the functionality of the circuit (e.g., [228]); and circuit equivalence checking in turn is sometimes done using circuit simulation (e.g., [71, 312]). Of these tasks, simulation is arguably the simplest, and is used in service of the other tasks; this makes it especially suited as a target application, and as a testbed for new techniques. In this thesis, therefore, we focus on quantum circuit simulation.

However, a major challenge in analyzing quantum algorithms using a classical computer is that the amount of memory required to describe an arbitrary quantum state grows exponentially in the number of qubits.[†] This is because, contrary to the classical world, where representing a system state consisting of m classical bits requires only a linear amount of memory, the state of an n-qubit quantum system is described by a vector of 2^n complex numbers, i.e., a vector in $\mathbb{C}^{2^n}$. Current estimates indicate that at least hundreds of qubits are required to perform useful tasks on a quantum computer [145]. However, even current super-computing clusters can only analyze systems with between 50 and 60 qubits represented as vectors [150, 173, 255].

Therefore, dedicated data structures and design methods which can tackle the exponential complexity of quantum computing need to be developed. By encoding quantum states using only a modest amount of memory, these data structures make quantum algorithm analysis tractable in many cases and thereby form the cornerstone on which many of these techniques are built. This motivates the main goal of this thesis: to analyze existing and design new dedicated data structures and apply them to the simulation and verification – specifically, equivalence checking – of quantum circuits.

As we will explain below, the various data structures we investigate in principle all support the same set of operations, namely, those operations necessary to simulate quantum circuits. Therefore, although in this thesis we apply our techniques primarily

---

[†] Specifically, a simple counting argument shows that any classical method requires at least $\Omega((1-\varepsilon)2^n)$ bits on average to describe arbitrary quantum states to a fidelity (informally, the "accuracy") of $1-\varepsilon$. See Section 2.2 for a definition of fidelity.

to the tasks of simulation and verification, we are optimistic that in the future they can be applied to quantum algorithm analysis more broadly and to other tasks to which these data structures have been applied, such as synthesis [6, 179, 279, 302, 347, 348, 373], uniform sampling and model counting [192, 195, 289], stochastic constraint optimization [99, 197, 198] and solving quantum many-body problems, such as those arising in condensed matter physics and quantum chemistry (e.g., [123, 250, 346] use matrix product states and tree tensor networks, and [78,132,225,236,247] use restricted Boltzmann machines; we will see some of these in Chapter 5).

Verification of software. In the formal verification of software, we wish to develop automated tools which guarantee that a given program works as intended. Formal verification goes beyond *testing*. Paraphrasing Dijkstra: testing can show the presence of bugs, but formal verification can show the absence of bugs [102]. Our aim will be to carry out the verification automatically, as opposed to pen-and-paper techniques (e.g., [205, 366]), and techniques in which a proof assistant, such as Isabelle [253, 254] or Coq [47,270], assists a human prover. In our setting, the advantage of an automated approach is that no human expertise is required for this step, so that these tools can be embedded in a toolchain and, e.g., process code alongside a compiler.

Today, formal verification is often an important part of the design process of software and hardware. Baier and Katoen [28] list many real-world examples (for example, the analysis of software of Rotterdam's storm surge barrier [316] and NASA's Deep Space 1 probe [151], among others).

The principal bottleneck in formal verification is the *state space explosion*: the simple observation that the number of states of a system grows exponentially with the size of the program. Among other things, this is caused by the number of different possible inputs, and the number of possible interleavings of parallel processes. Indeed, most approaches to verification have as their principal aim to somehow tame this state space explosion. They do so by exploiting structure in the system; for example, partial order reduction exploits the commutativity of (concurrent) processes [134,259,319], symbolic reachability analysis uses decision diagrams to exploit the structure of the reachable state space when viewed as a Boolean function [29, 70] and IC3 uses SAT solvers [60]. We will see that the effectiveness of verification methods for quantum software likewise hinges on their ability to harness the structure that is present in quantum states and systems. In fact, in Section 1.2 we will see that data structures previously used to address the state space explosion in classical systems are repurposed to represent quantum states.

Verification of quantum software. Our aim will be to verify whether two given quantum circuits are equivalent, i.e., whether they produce the same output state when given the same input state. This allows a user to, for example, check whether their (efficient but complicated) algorithm is equivalent to an (inefficient but obviously or provably correct) reference implementation. A quantum circuit on n qubits is described by a $2^n \times 2^n$ complex-valued matrix, so, concretely, checking the equivalence of two quantum circuits entails checking whether their matrices are equal.[‡]

This method is complementary but orthogonal to approaches, like model checking, in which one checks whether a (quantum) system satisfies a specification formulated in for instance a temporal logic. An advantage of our approach is that an equivalence checker can be integrated directly into a compiler or an optimizer (e.g., as in [228]) without requiring the user to supply a specification in a temporal logic. Therefore, in Section 2.4, we put these two approaches in context, highlighting similarities and differences. For now, we briefly mention that there are several quantum extensions of temporal logics [30,79,100,216,217,355,359,363,366,367], and several software tools provide support for these logics [16,18,116,128,161,205].

The challenge of verifying quantum software inherits the difficulties of the classical case and in addition poses several new ones, both practical and conceptual. We briefly list two difficulties below, drawing inspiration from Ying et al. [360,363] and Turrini [317].

1. A quantum state requires an exponential amount of data to describe in general. This is a practical problem which can be addressed by using data structures which can describe some states using only a modest amount of memory. We review existing data structures in Section 1.2, and describe new ones introduced in this thesis in Section 1.4.

2. A quantum algorithm admits an infinite number of possible inputs, which raises the conceptual challenge of inventing methods to guarantee that a quantum algorithm is correct on all input states. Burgholzer et al. articulate why enumerating and checking a property for all (finitely many) basis vectors is not sufficient [71]. We describe our chosen approach to this second difficulty in Section 1.4 and survey some existing approaches in Section 2.4.

[‡] In Sec. 2.2.1 we describe in more detail how such a matrix comes about. For the purposes of this introduction, it suffices to assume that, if we know the gates that constitute the circuit, then we can find the circuit's matrix.

To address the second difficulty, existing equivalence checkers have used two main approaches. Either the circuit is simulated on several randomly chosen input states (e.g., as in [71]), or the circuits' matrices are constructed and then we simply check whether the two matrices are equal[§] [74, 81, 162, 163, 241, 256, 345, 349, 356, 357]. Both approaches encounter the first difficulty listed above. Namely, during simulation, the intermediate states encountered are described by exponentially long vectors; and in the second approach, when building a circuit's matrix, such a matrix is exponentially large. Since the problem in both cases is that an object does not fit in memory, a natural solution is to use compression. Indeed, our solution of choice is to use data structures which compress either a quantum state vector, or a circuit's matrix. Therefore, we turn to this topic next.

1.2 State-of-the-art data structures for quantum algorithm analysis

The previous section argues that dedicated data structures are the key to effective quantum algorithm analysis. In this thesis, therefore, we investigate many popular state-of-the-art data structures that are used for the analysis of quantum algorithms, especially verification and simulation. These include the (extended) stabilizer formalism [3, 135], decision diagrams [211, 227, 331, 341, 374], matrix product states [248, 330, 346] and restricted Boltzmann machines [78, 174, 225]; data structures that we do not treat in thesis include tensor networks [284] and the ZX-calculus [89, 105, 256, 320], as well as SAT-based methods [34, 43, 351]. Each of these data structures can (i) represent the state of a quantum system at a given moment in time (among other things), and (ii) can simulate the execution of a quantum circuit on a given input.

The principal difficulty these methods are intended to address is the amount of memory required to represent an arbitrary quantum state. To this end, these methods aim to encode (i.e., compress) a quantum state using only a modest amount of computer memory by, broadly speaking, exploiting the structure that is often present in quantum states that arise in practice in quantum algorithms. Given such a compact representation, a quantum circuit can then be simulated "simply" by starting with a

---

[§] More precisely, we check whether two matrices are equal up to a complex constant; we explain this detail in Section 2.2 and sweep it under the rug for now.

data structure, e.g., a decision diagram (DD), representing the initial state and then repeatedly finding the next state by applying the next gate of the circuit to the current state, i.e., building a DD representing this next state. A given approach then succeeds in simulating a given circuit if the representations of the intermediate states fit in the available memory, which is achieved when, informally speaking, these intermediate states contain enough of the kind of structure that the data structure exploits. Comparing the degrees to which different data structures succeed in compressing a state, and simulating a circuit, is done using tools from *knowledge compilation*.

For the purposes of this introduction, let us first highlight two of these methods: the *stabilizer formalism* [3,135] and *decision diagrams* [227,374]. Afterward, we introduce knowledge compilation as a way to study the relative strengths of such data structures.

The stabilizer formalism. Stabilizer states are a subset of quantum states [135]. They play a fundamental role in quantum computing as they form the basis of many quantum protocols, such as communication protocols like superdense coding [42], cryptography [38,131,263], error-correcting codes [41,135,137,310] and measurement-based quantum computing [272]. Aaronson and Gottesman [3] note that, "Stabilizer states are expressive enough to encompass most "paradoxes" of quantum mechanics, including the GHZ experiment [138], quantum dense coding [42], and quantum teleportation [40]"; to this list we might add entanglement [298] and Bell's experiment [36]. Given their importance, it may be surprising that stabilizer states can be efficiently simulated on a classical computer. On the other hand, stabilizer states are not rich enough to encompass all of quantum computing, so this efficient simulation does not allow one to simulate arbitrary quantum circuits.

Stabilizer states are created by so-called stabilizer circuits, which are quantum circuits consisting of only a restricted gate set – the Clifford gates.[¶] Aaronson and Gottesman [3], Bravyi et al. [21, 61, 62, 63, 64] and Qassim et al. [268] show how to simulate a quantum circuit even when it contains non-Clifford gates. This *extended stabilizer formalism* can simulate any quantum circuit. This method expresses an arbitrary quantum state vector as a linear combination of stabilizer states. This way, a circuit on n qubits containing only k non-Clifford gates can be simulated in $\exp(k)\text{poly}(n)$ time, i.e., simulation is fixed-parameter tractable in the number of non-Clifford gates.[‖] Subsequently, Aranuchalam et al. [21] showed that even the more difficult problem

---

[¶] Stabilizer states, Clifford circuits and Clifford gates are defined in Sec. 2.2.3.

[‖] This bound can be improved if we consider a specific gate set. For example, a circuit containing only Clifford gates and k T-gates can be simulated in $\mathcal{O}(2^{\gamma k}\text{poly}(n))$ time with $\gamma \leq 0.228$ [62].

of checking whether two quantum circuits are equal is fixed-parameter tractable in the number of non-Clifford gates. The principal limitation of this method is that many interesting quantum circuits contain many non-Clifford gates. Moreover, even in theory, expressing a given quantum state as a linear combination of stabilizer states requires some minimum number of terms, called its stabilizer rank. Unfortunately, the stabilizer rank of interesting states may be expected to grow super-polynomially in the number of qubits on complexity-theoretic grounds,[**] although the current best explicit lower bounds are merely linear in the number of qubits [64, 209, 222, 260].[††]

Decision diagrams. A decision diagram (DD) aims to provide a compact representation of quantum state vectors (and quantum circuits) by exploiting the observation that a state vector often contains repeated subvectors, i.e., a state vector may contain multiple copies of the same block. By recognizing these repeated occurrences, a decision diagram can achieve lossless compression. This way, some state vectors can be represented very succinctly even for very large numbers of qubits. For example, any product state (i.e., a state without entanglement) on n qubits can be represented by certain DDs using only $\mathcal{O}(n)$ memory.

Decision diagrams were first introduced and developed by Lee [199], Akers [7] and Bryant [67] to analyze classical systems. Viamontes, Markov and Hayes first used a DD-based approach to simulate quantum circuits by introducing QuIDD, a decision diagram which can represent a vector of complex numbers, i.e., a quantum state [331, 333]. Miller and Thornton [227] introduced QMDDs, which specialize SLDDs [352] and AADDs [281] and which improve on QuIDDs by achieving greater compression. Since then, implementations of QMDDs and related DDs have achieved striking performance on a varied set of benchmarks, both for simulating quantum circuits [143, 156, 294, 368, 374] and for checking equivalence of quantum circuits [71, 74, 81, 162, 163, 241, 345, 349, 356, 357]. Based on empirical evaluations, Peham et al. conclude that "decision diagrams and the ZX-calculus can serve as complementary approaches" for quantum circuit equivalence checking [257].

Recent work has seen QMDDs applied to various subproblems of quantum simulation and verification and has improved many practical aspects of working with decision diagrams [142, 243, 294, 369, 370]. For example, Grurl et al. [141, 143, 144] and Hong

---

[**] For example, Morimae and Tamaki show that the stabilizer rank of the so-called magic state on n qubits grows as $2^{\Omega(n)}$ assuming the Exponential Time Hypothesis [232]. Mehraban and Tahmasbi show that the stabilizer rank of this magic state must grow super-polynomially, unless $\mathsf{MA} = \mathsf{P}^{\#\mathsf{P}}$ and the polynomial hierarchy collapses, which is widely thought to be unlikely [222].

[††] Recent work by Mehraban and Tahmasbi claims a quadratic lower bound [222].

et al. [162] develop DD-based techniques to address the challenges posed by noise and decoherence in quantum circuits. Second, Zulehner et al. [368] and Hillmich et al. [157] achieve better compression by periodically deleting parts of the diagram during simulation of a circuit, while guaranteeing that the fidelity (informally, the "accuracy") does not drop below a given threshold. Hong et al. use DD-based techniques to perform tensor network contraction, which is a general problem which includes quantum circuit simulation as a special case [163]. Lastly, multiple authors implement heuristics for dynamic variable reordering, which is a fundamental operation on DDs [156, 226]. Hillmich et al. find that variable reordering introduces challenges of numerical accuracy [156].

We highlight one QMDD-based method for circuit equivalence checking by Burgholzer et al. [71] in which the outputs of two given circuits are compared on random inputs. In this approach, QMDDs are used to simulate the two circuits on a randomly chosen input state; subsequently, we check whether the two circuits produced the same output state. The idea is that, if two circuits are different, then a random input likely reveals this. Specifically, the input states are random stabilizer states; this is necessary (as opposed to using simpler states, such as basis states, as inputs) in order to achieve good guarantees on the probability of finding such a counterexample.

Limitations of state-of-the-art tools for verification of quantum circuits. Early in our investigations, we discovered and proved that existing DDs cannot efficiently handle stabilizer states and Clifford circuits. This significantly limits their applicability to the analysis and verification of many common quantum protocols. Specifically, we proved the following about existing types of DDs:

> Existing types of decision diagrams require an exponential amount of memory to store some stabilizer states and (the unitary matrices of) some stabilizer circuits (Corollary 3.1). That is, stabilizer quantum circuits are a worst-case input for existing decision diagrams.

This may come as a surprise, because DDs are, ostensibly, designed to excel at analyzing states that possess structure, and stabilizer states certainly possess structure. This had been observed empirically earlier by Burgholzer et al. [71] for small numbers of qubits but had not been explained analytically. This discovery motivates our goal to find better decision diagrams, which are able to analyze stabilizer circuits in particular and common quantum protocols more broadly.

Although there are several available software tools for quantum verification, each suffers from certain limitations: those that are based on decision diagrams cannot efficiently handle stabilizer circuits (e.g., [71, 73, 75, 162]); some tools can handle only stabilizer circuits (e.g., [18, 128, 312]); other tools (e.g., [17, 43, 116, 161]) can handle arbitrary circuits but the chosen simulation method significantly limits the number of qubits the tool can handle (e.g., the tool of [43] generates SAT formulas that are exponentially large, except when analyzing stabilizer circuits).

Knowledge compilation. The approaches to verification that we study in this thesis each address the problem that quantum state descriptions are in general too large by using some data structure (DS) to (losslessly) compress a quantum state. These data structures are built on different architectures, which make some more suited to and more effective at certain tasks than others.

Knowledge compilation is the systematic study of such data structures in order to understand how effective a given DS is at a given task [97, 112, 113, 194]. To this end, researchers have primarily compared data structures on three criteria: *succinctness*, which tells us whether a data structure achieves good compression, and *tractability* and *rapidity*, which tell us how efficiently we can operate on the data in compressed form. A major goal of knowledge compilation is to help users choose a DS which is best suited to the task at hand. To this end, Darwiche and Marquis [97] formulate the following advice: when considering which DS to use, one should choose the most succinct DS which performs the query of interest in polynomial time.

However, the tools of knowledge compilation have not yet been systematically applied to data structures used for analysis of quantum computing. We now sketch four shortcomings of the literature on this topic.

First, the succinctness relations between popular data structures were unknown, e.g., it was not known whether matrix product states were more succinct than QMDDs. Therefore, it is not obvious how a practitioner should follow Darwiche's advice, which relies on the succinctness relations.

Second, it is not obvious which operations are important for such data structures in the context of quantum algorithm analysis (i.e., in terms of tractability and rapidity). For example, it seems useful that a data structure be able to efficiently compute the probability of a given measurement outcome for a given quantum state. However, tractability of one operation (such as measurement) is sometimes traded off against another operation (such as applying certain gates). Although for several data structures efficient algorithms are known for some of these operations, to the best of our knowledge there has been no systematic comparison of which tradeoffs are available, or how to prioritize such tradeoffs. Moreover, the traditional operations considered in knowledge compilation (e.g., in [97, 112, 194]), such as conjunction, disjunction and existential quantification, are inherently Boolean operations. However, quantum state vectors are not Boolean but complex vectors, so it is not obvious what would be natural quantum analogues for these operations. For example, Fargier et al. [113] consider the tractability of *pointwise addition* of two vectors, which in principle can be applied to complex-valued vectors such as quantum state vectors; however, the addition of two quantum states is not a meaningful quantum operation.

Third, the rapidity criterion, introduced by Lai et al. [194], is defined only for *canonical* data structures, i.e., for those data structures which provide a unique normal form for each quantum state. However, many data structures in popular use for quantum algorithm analysis are not canonical, such as restricted Boltzmann machines and matrix product states. Therefore, the criterion cannot be applied to these data structures, which limits our ability to compare them. Consequently, we cannot say that, e.g., matrix product states are more rapid than QMDDs, because MPS are not a canonical data structure.

Fourth, considering only succinctness and tractability sometimes leads to apparently incongruent results. For example, representing an n-(qu)bit state vector using an explicit vector representation always uses $\mathcal{O}(2^n)$ memory, whereas using a QMDD uses much less memory in some cases. However, applying a certain quantum gate called a Hadamard gate to a quantum state is considered tractable for the vector representation (because it can be performed in time polynomial in the length of the vector) but is considered intractable for the QMDD representation. This is counterintuitive: using QMDDs for this task is never much slower than using the state vector representation, and indeed is sometimes much faster. Therefore, it appears as though there is a tradeoff between succinctness and tractability in this case, but it is not obvious whether the implied tradeoff between speed and memory is real; or to put it more mildly, it is not clear what conclusions one should draw from this data point.

1.3 Research questions

In the previous section, we have sketched the current frontier of research into quantum verification methods. To some, it may come as a surprise that analyzing quantum computers on classical hardware is possible even in principle, but in fact, as we have seen above, there are today a variety of methods and tools to verify the correctness of quantum algorithms. However, these tools are still in their infancy relative to the mature field of classical verification. For example, several data structures, such as matrix product states and restricted Boltzmann machines, have not yet been applied to the verification of quantum algorithms to the best of our knowledge. Moreover, the limitations of several state-of-the-art methods, such as decision diagrams and the stabilizer formalism, leave opportunities for improvement. To push this frontier forward, we now ask three research questions which will guide the work in this thesis.

First, we seek to address the limitations of decision diagrams and the stabilizer formalism. In particular, we have remarked that stabilizer circuits are a worst-case scenario for existing decision diagrams because they need exponential size to represent most stabilizer states. Therefore, we ask:

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

We would consider this question answered if we found a data structure which incorporates the strengths of the stabilizer formalism and of existing decision diagrams.

We have argued that the existing literature on knowledge compilation insufficiently covers the case of quantum algorithm analysis. In particular, the data structures in popular usage have not yet been compared on their succinctness, many non-canonical data structures cannot be compared on their rapidity and it is not clear how to prioritize the tractability of operations when designing new data structures. Therefore, we ask:

> **Research question 2.** *How can we analytically compare the relative strengths of data structures which represent quantum states?*

To date, not all the different DDs have been adapted to the quantum setting.[‡‡] For

---

[‡‡] A DD encodes a function $f\colon \{0,1\}^n \to R$ for some range R; we say a DD is *classical* if $R \subseteq [0,1]$

example, there are no quantum SDDs [96], DSDBDDs [46, 264], FBDDs [129], or ZDDs [229]. Given that DDs have been widely applied to and have shown striking success in the verification of classical software, it is interesting to ask which DDs would be effective in this new setting, if they were suitably adapted.

> **Research question 3.** *Which classical decision diagrams might be effective for the analysis of quantum algorithms, if they were suitably adapted?*

1.4 Contributions

In light of the bottlenecks of current state-of-the-art methods described above in Section 1.2, we introduce a novel data structure called *Local Invertible Map Decision Diagram* (LIMDD), which unites the strengths of DDs and the stabilizer formalism, in Chapter 3, thus addressing Research question 1. Specifically, LIMDDs can efficiently analyze both (i) all stabilizer circuits, and (ii) all circuits that existing DD-based tools can analyze, whereas existing DD-based tools required exponential resources to analyze stabilizer circuits. In fact, LIMDDs can efficiently analyze some non-stabilizer circuits that existing DD-based tools cannot efficiently analyze; therefore, LIMDDs are more than merely the "union" of existing DDs with the stabilizer formalism. In general, LIMDDs can simulate any quantum circuit (regardless of the gate set) on any initial state; and they can represent (the unitary matrix of) any quantum circuit. Many important quantum operations take only polynomial time in the size of the LIMDD, such as checking the equality of two quantum circuits, computing the probability of obtaining a given measurement outcome, sampling from a quantum state, and applying various gates, such as the (controlled) Pauli X, Y, Z gates. We explain Burgholzer et al.'s [71] empirical finding that QMDDs are unable to efficiently handle stabilizer states by proving that there are stabilizer states which can only be represented by exponentially-sized QMDDs. In the other direction, we show that QMDDs can represent quantum states that are not stabilizer states. Next, we compare LIMDDs to the extended stabilizer formalism and find that we have an exponential advantage over a popular instantiation of this formalism, which we call the Clifford+T simulator, assuming the exponential time hypothesis.

We implement LIMDDs in the publicly available software package MQT [233]. In

---

and *quantum* if $R = \mathbb{C}$.

Chapter 4, we show empirically that we are able to make good on the promises made in Chapter 3: our tool outperforms a state-of-the-art method when analyzing quantum circuits that implement the Quantum Fourier Transform (QFT). The QFT is a subroutine of many quantum algorithms, so it is important that proposed circuits are correct. We use the method of Burgholzer et al. [71], who check the equivalence of two quantum circuits by simulating them on a set of randomly chosen input states. The idea is that if two circuits are different, then one may expect that a random input will reveal this, since the outputs of the circuits will differ with high probability.

To answer Research question 2, we give a quantum knowledge compilation map in Chapter 5. We map the succinctness, tractability and rapidity of some of the currently popular data structures for analyzing quantum systems: decision diagrams, matrix product states and restricted Boltzmann machines. We show that LIMDDs and matrix product states are more rapid than QMDDs, which in turn are more rapid than the state vector representation. We argue that the most important operations in the context of quantum algorithm analysis are measurement, computing inner products, and applying various common quantum gates; we investigate the tractability of these operations for the data structures listed above. Lastly, we contribute to the *methodology* of knowledge compilation (i) by extending the definition of rapidity by removing the constraint that the data structures are canonical, which allows us to compare data structures that previously could not be compared; (ii) by giving a simple sufficient condition for when one (non-canonical) data structure is more rapid than another; (iii) by establishing the rapidity relations between various data structures, thus showcasing that the condition is simple to use and (iv) more broadly, by making a knowledge compilation map involving, for the first time, data structures that are not a variant of a so-called Negation Normal Form structure (see [33, 94]).

In Research question 3, we ask which classical decision diagrams might contribute to the analysis of quantum algorithms. To this end, we investigate two DDs: the Sentential Decision Diagram (SDD) [96] and the Disjoint Support Decomposition Binary Decision Diagram (DSDBDD) [46, 264]. For the time being, we test them out in the classical domain. It is instructive to do this, because the quantum setting poses several additional challenges relative to the classical setting. Notably, a quantum diagram needs to account for complex numbers, adding complexity to its architecture. Moreover, the finite precision of arithmetic leads to numerical inaccuracies and to questions of numerical stability. Meeting these challenges entails increasing the conceptual, and sometimes computational, complexity of the algorithms. For example, variable reordering is a fundamental and conceptually simple task for DDs. However, Hillmich et al. [156] find that this task becomes significantly error-prone for quantum DDs due to rounding errors that are inherent to finite precision floating-point arithmetic. By deferring these challenges to future work, we obtain diagrams which are conceptually simpler. This allows us to go through more prototyping cycles, e.g., in our case, for testing heuristics for setting an SDD's hyperparameters (specifically, its *variable tree*). We speculate about how to extend these two DDs to the quantum domain in Section 8.3.

In this context of future diagrams, we contribute the following two results. First, in Chapter 6, we prove that the DSDBDD sometimes achieves exponentially better compression than many other DDs. This solves succinctness but leaves unsolved the questions of tractability and rapidity.

Second, in Chapter 7, we extend the LTSmin model checking framework with SDDs, so that users can take advantage of the better compression that it offers over other DDs that were previously integrated with LTSmin. Effectively deploying SDDs requires us to propose the first heuristics for SDDs: we design the structure of a certain binary tree (a "variable tree") based on limited available information about which variables are read and written by the software program that we are trying to verify. Our experiments show that SDDs allow us to make a well-known tradeoff between time and space: the SDDs are slower than BDDs, but they achieve better compression; therefore, a patient user may find that these DDs allow her to solve larger problems without running out of memory.

Although we have pitched these two chapters as preparing the way for quantum applications, we note that these contributions may be valuable in their own right. For example, users of the LTSmin toolset can already use SDDs, regardless of their future applications to quantum computing.

1.5 Outline

The chapters of this thesis can be summarized as follows.

Chapter 2 We give the necessary background on quantum computing, decision diagrams and the stabilizer formalism.

Table 1.1: The chapters of this thesis and their methods.

Chapter	Domain	Method	Data structure	Artefact
Chapter 3	Quantum	Analytical	LIMDD	
Chapter 4	Quantum	Empirical	LIMDD	MQT [233]
Chapter 5	Quantum	Analytical	Various	
Chapter 6	Classical	Analytical	DSDBDD	
Chapter 7	Classical	Empirical	SDD	LTSmin [210]

Chapter 3 We introduce the LIMDD data structure, give algorithms for quantum simulation using this data structure and prove that it is faster than matrix product states and the extended stabilizer formalism for simulating certain quantum circuits.

Chapter 4 We describe our implementation of the LIMDD. We evaluate our implementation empirically by analyzing a circuit which implements Quantum Fourier Transform.

Chapter 5 We provide a knowledge compilation map for quantum computing: we compare five data structures on their succinctness, tractability and rapidity and we give a sufficient condition for one data structure to be more rapid than another.

Chapter 6 We prove that DSDBDDs are exponentially more succinct than BDDs.

Chapter 7 We use SDDs to do model checking on a large standard problem set. To this end, we extend the LTSmin package to provide support for SDDs.

Chapter 8 We conclude by reflecting on the work in this thesis and laying out some open problems for future research.

Table 1.1 lists, for each chapter in this thesis, its main data structure, the method by which it analyzes it and a link to the resulting software package, if any.

The proofs of many theorems appear in the appendices.

1.6 Publications and artefacts

This dissertation is based on the following publications.

1. **Lieuwe Vinkhuijzen** and Alfons Laarman. Symbolic Model Checking with Sentential Decision Diagrams. International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. Springer, Cham, 2020.
2. **Lieuwe Vinkhuijzen** and Alfons Laarman. The Power of Disjoint Support Decompositions in Decision Diagrams. NASA Formal Methods Symposium. Springer, Cham, 2022.
3. **Lieuwe Vinkhuijzen**, Tim Coopmans, David Elkouss, Vedran Dunjko and Alfons Laarman. LIMDD a decision diagram for simulation of quantum computing including stabilizer states. Quantum, 2023.
4. **Lieuwe Vinkhuijzen**, Thomas Grurl, Stefan Hillmich, Sebastiaan Brand, Robert Wille and Alfons Laarman. Efficient implementation of LIMDDs for Quantum Circuit Simulation. 29th International Symposium on Model Checking of Software, 2023.
5. **Lieuwe Vinkhuijzen**, Tim Coopmans and Alfons Laarman. A Quantum Knowledge Compilation Map. arXiv preprint, arXiv:2401.01322 (2024).

This dissertation features contributions to the following software packages.

1. **LIMDD.** We implemented the LIMDD in the MQT DDSIM package, allowing a user to simulate arbitrary quantum circuits [233].
2. **LTSmin.** We augmented the LTSmin package, giving the user the option to use Sentential Decision Diagrams [210]. This software package forms the basis of Chapter 7.

Besides the publications included in this thesis, we co-authored the following paper.

1. **Lieuwe Vinkhuijzen** and Andre Deutz. A simple Proof of Vyalyi's Theorem and Generalizations. Electronic Colloquim on Computational Complexity, 2019.

Chapter 2

Preliminaries

This chapter covers the background necessary for the remainder of the thesis: we introduce the basic notion of quantum computing in Section 2.2; we introduce decision diagrams, which are the primary data structures considered in this thesis, in Section 2.3; we introduce some methods for comparing data structures (in Sec. 2.3.2); lastly, we provide a brief overview of quantum model checking in Section 2.4. Each chapter contains in addition a preliminary section highlighting material for that chapter. The reader may opt to skip this chapter and instead read the specific preliminaries found in the relevant chapters.

Further reading. The interested reader is encouraged to consult the books of Nielsen and Chuang, [240] or of Kitaev, Shen and Vyalyi [182] on quantum information and computing. Wille and Zulehner's book [376] lays out the problem of design automation for quantum computing, and explains decision diagrams as a method for addressing this problem. Wegener's book gives a more broad view on decision diagrams [344]. A basic understanding of complexity theory, such as can be found in the first few chapters of the books of Papadimitriou [252] and Arora and Barak [20], will be useful to understand the analysis of our algorithms and several hardness proofs in this thesis. By far the most amusing text which covers much of the above material is Aaronson's *Quantum Computing Since Democritus* [2].

The following resources are complementary but somewhat tangential to the topics in this thesis. A good introduction to model checking can be found in Baier and Katoen [28], or in the more recent book of Clarke et al. [88]. For quantum model checking, one may consult the book by Ying and Feng [361], or their short survey on the topic [360]. Turrini provides an accessible introduction to the topic [317]. Garwahl et al. [126] and Ferreina [118] provide excellent surveys of quantum programming languages.

2.1 Mathematical preliminaries

A *Boolean function* is a map $f\colon \{0,1\}^n \to \{0,1\}$ which associates each bit-string $x \in \{0,1\}^n$ with a bit $f(x) \in \{0,1\}$. The bits that make up this bit-string are the *input variables* and will be denoted by lower case letters, for example as $x_1, \dots, x_n$. If $a \in \{0,1\}^\ell$ is a bit-string of length ℓ, and if $\ell \leq n$, then a *induces the subfunction* $f_a\colon \{0,1\}^{n-\ell} \to \{0,1\}$. This induced subfunction is defined by $f_a(y) = f(a,y)$ for $y \in \{0,1\}^{n-\ell}$ (sometimes we say that f_a *restricts* f to a). The string a is called a *partial assignment* to the variables of x.

Example 2.1. Consider the formula $x_1 \vee x_2$. This formula has an associated function $f\colon \{0,1\}^2 \to \{0,1\}$, which takes values $f(x_1,x_2) = x_1 \vee x_2$. This function outputs the value $f(x_1,x_2) = 0$ at the point $(x_1,x_2) = (0,0)$ and outputs $f(x_1,x_2) = 1$ everywhere else. Consider now the length-1 bit-string $a = 0$. This partial assignments induces the subfunction $f_a\colon \{0,1\}^1 \to \{0,1\}$. This function takes the values $f_a(x_2) = x_2$ for $x_2 \in \{0,1\}$.

This example also teaches us that, although a formula is associated with a function, the two are not the same thing; notably, two different formulas may effect the same function, e.g., the formula $x_1 \vee x_2$ has the same truth values as the formula $x_1 \vee (x_1 \wedge x_2)$.

We will often make use of the *Shannon decomposition* of a function [288]. For any function f, and a single Boolean variable x_1, its Shannon decomposition with respect to x_1 is the following expression,

$$f(x_1, x_2, \dots, x_n) = \neg x_1 \wedge f_0(x_2, \dots, x_n) \;\vee\; x_1 \wedge f_1(x_2, \dots, x_n) \tag{2.1}$$

This construction can be applied recursively to the $n-1$-variable functions f_0 and f_1. The usefulness of the Shannon decomposition lies in that it allows us both to break down a function into its constituent parts (as is done in Equation 2.1); and to build a function from smaller functions on fewer variables (namely, in a scenario in which we have two functions f_0 and f_1, we can use Equation 2.1 to build the larger function f).

A function of the form $f\colon \{0,1\}^n \to \mathbb{C}$ is called a *pseudo-Boolean* function. The concepts of induced subfunctions and Shannon decompositions are applicable here just as to Boolean functions.

For a set A of group elements, we write $\langle A\rangle$ to denote the group generated under multiplication. For example, consider a set of two matrices, $A = \{\left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right], \left[\begin{smallmatrix}-1 & 0\\ 0 & 1\end{smallmatrix}\right]\}$. The group $\langle A\rangle$ generated by this set is

$$\langle A\rangle = \left\{\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right], \left[\begin{smallmatrix}1 & 0\\ 0 & -1\end{smallmatrix}\right], \left[\begin{smallmatrix}-1 & 0\\ 0 & 1\end{smallmatrix}\right], \left[\begin{smallmatrix}-1 & 0\\ 0 & 1\end{smallmatrix}\right]\right\} \tag{2.2}$$

Given an $n \times m$ matrix A, and a $k \times \ell$ matrix B, the *tensor product* of matrices A and B, denoted $A \otimes B$, is the following $nk \times m\ell$ matrix,

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1m}B \\ a_{21}B & \ddots & \cdots & a_{2m}B \\ \vdots & \vdots & \ddots & \cdots \\ a_{n1}B & a_{n2}B & \cdots & a_{nm}B \end{bmatrix} \tag{2.3}$$

Note that each entry $a_{ij}B$ above denotes a $k \times \ell$ matrix block. The tensor product is associative but not commutative. Notably, it distributes over the usual matrix product: $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$. We will often consider the special case of the tensor product when A and B are both vectors. We will write $A^{\otimes k} = A \otimes \cdots \otimes A$ to denote the k-fold tensor product of a matrix A.

2.2 Introduction to quantum computing

The basic unit of information in quantum computers is the quantum bit, called a *qubit*.[*] The state of a qubit (at a specific moment in time) is described by a complex vector $[\,\alpha_0,\ \alpha_1\,]^T \in \mathbb{C}^2$ with norm 1, usually written in Dirac notation[†] as $\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$.

---

[*] A qubit can be physically realized in many different ways, e.g., as the polarization of a photon, or as the spin of an electron. By analogy, a classical bit can be described by the abstract entities 0 or 1, independent of how the bit is physically realized, e.g., stored on a magnetic tape, or on a cd, or as an electric charge in a DRAM, or even as several copies of one of the above in an error-corrected context. Likewise, in this thesis, we will deal with qubits only abstractly as vectors in a complex Hilbert space, which is independent of their physical realization in a quantum computer.

[†] Aaronson notes, "This notation usually drives computer scientists up a wall when they first see it – especially because of the asymmetric brackets! But if you stick with it, you see that it's really not so bad." [2]

Here $|0\rangle$ is shorthand for the vector $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $|1\rangle$ is shorthand for $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$, so the expression above is shorthand for

$$\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle = \alpha_0 \cdot \begin{bmatrix}1\\0\end{bmatrix} + \alpha_1 \cdot \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\alpha_0\\0\end{bmatrix} + \begin{bmatrix}0\\\alpha_1\end{bmatrix} = \begin{bmatrix}\alpha_0\\\alpha_1\end{bmatrix} \tag{2.4}$$

and satisfies

$$|\alpha_0|^2 + |\alpha_1|^2 = 1 \qquad \textbf{i.e., quantum state vectors have norm } 1 \tag{2.5}$$

The complex numbers α_0, α_1 are called the *amplitudes* of the state. Such an amplitude vector describes all information about the quantum state. If both amplitudes α_0 and α_1 are non-zero, the state is in *superposition*. It follows from Equation 2.5 that the zero vector is not a quantum state.

It is sometimes convenient, when doing calculations, to temporarily neglect the constraint that quantum state vectors have norm 1 (e.g., when doing calculations by hand, or when using methods such as QMDDs). A quantum state vector is called *(non-)normalized* when it (does not) have norm 1, to emphasize which convention is used.

Example 2.2. The states $|0\rangle = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \in \mathbb{C}^2$ and $|1\rangle = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ are (normalized) quantum states on one qubit. The state $|+\rangle = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$ is in superposition of $|0\rangle$ and $|1\rangle$.

Two amplitude vectors differing only by a complex factor are considered (physically) equivalent. More precisely, if $|\varphi\rangle, |\psi\rangle$ are quantum state vectors, and $|\varphi\rangle = \lambda|\psi\rangle$ for some complex value $\lambda \in \mathbb{C}$, then $|\varphi\rangle$ and $|\psi\rangle$ are said to describe the same quantum state. Notably, no physical measurement can distinguish between two such equivalent amplitude vectors.

A quantum register may consist of multiple qubits. A quantum register consisting of n qubits has 2^n basis states $|x\rangle$ with $x \in \{0,1\}^n$, each with a corresponding amplitude $\alpha_x \in \mathbb{C}$. We denote $|x\rangle = |x_n\rangle \otimes \cdots \otimes |x_1\rangle = |x_n \ldots x_1\rangle$, for example, $|00\rangle = |0\rangle \otimes |0\rangle$. The state $|\varphi\rangle$ can therefore be written,

$$|\varphi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle \tag{2.6}$$

The normalization constraint, requiring $|\varphi\rangle$ to be a unit vector, is generalized to:

$$\sum_{0 \leq x < 2^n} |\alpha_x|^2 = 1 \tag{2.7}$$

A quantum state $|\varphi\rangle$ is said to be *entangled* if it cannot be written as a tensor product of single qubit states, i.e, as $|\varphi\rangle = |\varphi_1\rangle \otimes \cdots \otimes |\varphi_n\rangle$.

Example 2.3. Consider the quantum state $1/\sqrt{2} \cdot (|00\rangle + |11\rangle)$, known as the *Bell state* [240]. As a vector, it would be written as $1/\sqrt{2} \cdot [\,1\ 0\ 0\ 1\,]^{\mathrm{T}}$. In addition to superposition, this quantum state shows *entanglement.*

Alternatively, a state $|\varphi\rangle$ can be understood as the pseudo-Boolean function $f\colon \{0,1\}^n \to \mathbb{C}$, where $f(x) = \alpha_x$. It follows that an amplitude vector can also be expressed using a Shannon decomposition, as follows,

$$|\varphi\rangle = \alpha_0 |0\rangle \otimes |\varphi_0\rangle + \alpha_1 |1\rangle \otimes |\varphi_1\rangle \tag{2.8}$$

where $|\varphi_0\rangle, |\varphi_1\rangle$ are $n-1$-qubit states and $\alpha_0, \alpha_1 \in \mathbb{C}$. Put simply, this expression defines the vector $|\varphi\rangle$ as the vector whose top half is $\alpha_0 \cdot |\varphi_0\rangle$ and whose bottom half is $\alpha_1 \cdot |\varphi_1\rangle$.

We denote the complex conjugate transpose of a given vector $|\varphi\rangle$ as $\langle\varphi| = (|\varphi\rangle^T)^\dagger$. Thus, Dirac notation makes it easy to see that $|\varphi\rangle$ is a column vector and $\langle\varphi|$ is a row vector. It follows, for example, that $\langle\varphi| \cdot |\psi\rangle$ (or $\langle\varphi|\psi\rangle$ for short) is a scalar. A column vector $|\varphi\rangle$ is called a *ket* (or *ket vector*); a row vector $\langle\varphi|$ is called a *bra*; for this reason Dirac notation is also called *bra-ket notation.* For example, if $|\varphi\rangle = \frac{3}{5}|0\rangle + i\frac{4}{5}|1\rangle$, then its complex conjugate transpose is,

$$\langle\varphi| = \frac{3}{5}\langle 0| - i\frac{4}{5}\langle 1| = \begin{bmatrix} \frac{3}{5} & -i\frac{4}{5} \end{bmatrix}^T \tag{2.9}$$

The value $|\langle\varphi|\psi\rangle|^2$ is the *fidelity* of $|\varphi\rangle$ and $|\psi\rangle$, and tells us how close two states are.

Measurement of quantum states. A quantum state can be *measured.* For the purposes of this thesis, it will be sufficient to consider only single-qubit measurements in the computational basis.[‡] Suppose that a quantum state $|\varphi\rangle$ can be written as in Equation 2.8 and that $|\varphi_0\rangle, |\varphi_1\rangle$ are normalized vectors, i.e., with norm 1. When the

---

[‡] For a broader treatment of measurement in quantum computing, see Nielsen and Chuang [240].

first qubit of this state is measured, this results in either outcome 0, or outcome 1. The probability of obtaining outcome m is equal to

$$\mathbb{P}[\text{measurement outcome is } m] = \frac{|\alpha_m|^2}{|\alpha_0|^2 + |\alpha_1|^2} \tag{2.10}$$

After measurement, the state “collapses”. Specifically, if the outcome 0 was measured, then the state after measurement (the *post-measurement state*) is $|0\rangle \otimes |\varphi_0\rangle$; similarly, if 1 was measured then the state becomes $|1\rangle \otimes |\varphi_1\rangle$. In the new state, the first qubit is no longer entangled with the other qubits. This is reflected in the amplitude vector: if, for example, the measurement outcome was $m = 1$, then the first 2^{n-1} entries of the state vector are set to 0. For this reason, measurement is sometimes called *destructive*, since information about the quantum state is lost.

Example 2.4. Consider the Bell state from Example 2.3. If this state is measured, then the post-measurement state has equal probabilities of being one of the basis states $|00\rangle$ and $|11\rangle$, and zero probability of seeing the states $|01\rangle$ and $|10\rangle$. Measuring a value for one qubit of the Bell state immediately fixes the value of the other qubit corresponding to the measurement outcome, e.g., after measuring $q_1 = |0\rangle$ (or $q_1 = |1\rangle$) we immediately know that $q_0 = |0\rangle$ (or $q_0 = |1\rangle$).

2.2.1 Quantum states and operations

Quantum states are manipulated by applying quantum gates. A quantum gate is any unitary linear operator, i.e., a linear operator mapping quantum states to quantum states. A gate on n qubits, therefore, corresponds to a matrix $U \in \mathbb{C}^{2^n \times 2^n}$. If a quantum state $|\varphi\rangle$ serves as input to a gate U, then the output is the quantum state $U \cdot |\varphi\rangle$. A *quantum circuit* consists of a series of gates applied sequentially. Therefore, if an circuit U consists of sequentially applying first the gate U_1, then U_2, etc., until U_m, then the action of the circuit U is described by the $2^n \times 2^n$ unitary matrix $U_m \cdots U_1$, i.e., the matrix of U is simply the product of the (matrices corresponding to the) gates. We say that a quantum circuit $U_1, \ldots, U_m$ is equivalent to another quantum circuit $V_1, \ldots, V_\ell$ iff they effect the same unitary matrix modulo a complex factor, i.e., if $U_m \cdots U_1 = \lambda \cdot V_\ell \cdots V_1$ for some $\lambda \in \mathbb{C}$. The reason for considering operators up to a complex factor is because two unitary operators U, V map equivalent quantum states to equivalent quantum states if and only if these unitaries are equal modulo a complex factor, i.e., if $U = \lambda V$.

If U is a k-qubit gate, and it is applied to the first k qubits of a quantum register containing n qubits, then its action on the n-qubit system is described by the unitary matrix $U \otimes \mathbb{I}^{\otimes n-k}$. Here $\mathbb{I} = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$ is the identity gate and $\mathbb{I}^{\otimes n-k} = \mathbb{I} \otimes \cdots \otimes \mathbb{I}$ is its $(n-k)$-fold tensor product; this is a matrix of size $2^{n-k} \times 2^{n-k}$. We say that U *acts as the identity* on the remaining $n-k$ qubits. More generally, a gate U may of course be applied to any subset of qubits, not only to the first qubits in some variable order, as above. A matrix which acts as the identity on all but k qubits, such as the matrix $U \otimes \mathbb{I}^{\otimes n-k}$ above, is called *k-local*; in particular the gate U is also called k-local (we use the term *local* independent of whether the affected qubits are located next to each other).

If U and V are two quantum gates, then the matrix $U \otimes V$ applies U and V simultaneously to separate quantum registers. This is called the *parallel composition* of two quantum gates. The earlier expression $U \otimes \mathbb{I}^{\otimes n-k}$ can be thought of as a special case of parallel composition: we apply U to a k-qubit register, and apply the identity gate $\mathbb{I}$ to the remaining $n-k$ qubits.

Example 2.5. Three examples of common quantum gates are the single-qubit phase-shift operation S, the single-qubit Hadamard operation H, and the two-qubit controlled-NOT operation *CNOT* (here shown with control on the first qubit, target on the second qubit).

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad \mathit{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.11}$$

Example 2.6. The Bell state from Example 2.3 is created by the circuit in Figure 2.1. In this circuit, the quantum state starts in the initial state $|00\rangle$. First, a Hadamard gate is applied to the first qubit. The corresponding unitary is $H \otimes \mathbb{I}$, so the state after applying this gate is

$$H \otimes \mathbb{I} \cdot |00\rangle = \frac{1}{\sqrt{2}}(|1\rangle + |0\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \tag{2.12}$$

Next, a controlled-NOT gate is applied. Here the the first qubit acts as the *control*, denoted in the circuit by a •; the second qubit is the *target*. After applying the

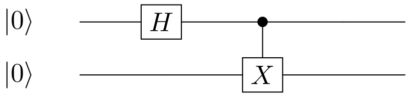


Figure 2.1: A 2-qubit quantum circuit preparing the Bell state from Example 2.3. It contains two gates: first a Hadamard is applied to the first qubit, then a controlled X gate (a $CNOT$ gate) is applied. The top qubit is the *control* and the bottom qubit is the *target* of this controlled X gate.

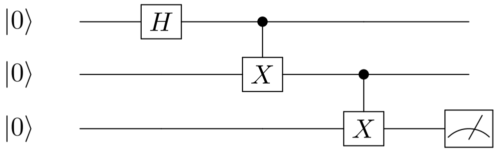


Figure 2.2: 3-qubit quantum circuit preparing the GHZ state. The rightmost box denotes a measurement to be performed on the last qubit.

controlled-NOT gate, the state is

$$\mathrm{CNOT} \cdot \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.13}$$

The circuit is described by the 4×4 matrix $U = CNOT \cdot (H \otimes \mathbb{I})$.

Example 2.7. Figure 2.2 gives an example of a circuit preparing a quantum state on 3 qubits, the so-called GHZ state: $|\mathrm{GHZ}\rangle = {}^1/\sqrt{2}(|000\rangle + |111\rangle)$. At the end of the circuit, after the three gates are applied, the last qubit is measured. This measurement has equal probability of measuring a 0, collapsing the state to $|000\rangle$, or measuring a 1, collapsing the state to $|111\rangle$.

As the examples above illustrate, a quantum circuit is typically given as a sequence of local gates (as opposed to as a sequence of exponentially large matrices). Consequently, a circuit can be succinctly described, even if the number of qubits is large.

Table 2.1 lists the most important gates encountered in this thesis. It groups gates into *Pauli gates* and *Clifford gates*, which we introduce shortly, in Sec. 2.2.3.

2.2.2 How to simulate a quantum circuit

Given a quantum circuit $U_1, \ldots, U_m$ and an initial state $|\varphi_0\rangle$, this circuit can be simulated on a classical computer straightforwardly by repeated matrix-vector multiplication. The simulation starts with the initial state and applies the quantum gates one after the other, obtaining the intermediate states $|\varphi_1\rangle, \ldots, |\varphi_m\rangle$. Each quantum operation is represented by a unitary matrix U_t of dimension $2^n \times 2^n$ and each quantum state by a unit vector $|\varphi_t\rangle$ of dimension 2^n (with $|\varphi_0\rangle$ commonly chosen to be $|\varphi_0\rangle = |0 \ldots 0\rangle$, called the all-zero state). The evolution of a state at time step t is then given by $|\varphi_{t+1}\rangle = U_{t+1} |\varphi_t\rangle$. In principle, these matrices and vectors can be stored in memory in a straightforward way as matrices of complex numbers. However, the memory requirements of this straightforward approach grow exponentially with the number of qubits (namely as 2^n for the vectors and 4^n for the matrices). For many quantum circuits, the simulation can be conducted much more efficiently by employing data structures to compactly store the intermediate states, such as the ones we introduce in Section 2.3.

2.2.3 Clifford circuits, stabilizer states and Pauli operators

An important subset of quantum circuits are Clifford circuits [137], which consist only of the three Clifford gates S, H and $CNOT$, defined in Equation 2.11. Clifford circuits play an essential role in many quantum subroutines and protocols, such as superdense coding [42], cryptography [38, 131, 263], error-correcting codes [41, 135, 137, 310] and measurement-based quantum computing [272] Clifford circuits can be efficiently simulated on a classical computer [3]. We remark that this does not immediately give an efficient way to simulate all quantum circuits, because Clifford circuits do not capture all of quantum computing: some quantum circuits cannot be expressed using only Clifford gates (see, e.g., [61, 63]). Table 2.1 lists some common Clifford gates.

We briefly sketch the idea of fast simulation of Clifford circuits before we give a detailed exposition. The fast simulation algorithm for Clifford circuits is based on Gottesman and Knill's [136] description of quantum states based on symmetries, rather than (exponential) amplitude vectors. In this formalism, we consider a set of operators which *stabilize* a state (specifically, we consider the Pauli operators that stabilize a state; see below for a definition). This set is called the *stabilizer group* of that state. The idea is to uniquely identify a state with its stabilizer group. The set of states that can be described this way is called the *stabilizer states.* A state's stabilizer group can be efficiently represented by its generator set, which is what yields a representation of a set of states that is more compact than an amplitude vector. Lastly, consider a state $|\varphi\rangle$ with stabilizer group G, and a Clifford gate U, so that the state after applying the gate is $U|\varphi\rangle$. Then the stabilizer group of $U|\varphi\rangle$ can be efficiently computed given G and U. The same is true if we wish to simulate a measurement: the probability of obtaining a given measurement outcome can be efficiently computed, and stabilizer group of the post-measurement state can be efficiently found. This yields the fast simulation algorithm. A compact representation is feasible because there are only few n-qubit stabilizers for any $n \geq 1$, namely, $2^{\mathcal{O}(n^2)}$. The relation between Clifford circuits and stabilizer states is that the set of stabilizer states is precisely the set of intermediate and final states of Clifford circuits, if the initial state was the all-zero state $|0\rangle$.

We now describe the stabilizer formalism in detail. A unitary operator U *stabilizes* a state $|\varphi\rangle$ if $|\varphi\rangle$ is a $+1$ eigenvector of U, i.e., if it satisfies $U|\varphi\rangle = |\varphi\rangle$. The set of stabilizers of any state $|\varphi\rangle$ forms a group, since if U and V stabilize $|\varphi\rangle$, then so do UV, VU and $U^\dagger$. In the stabilizer formalism, we consider only those stabilizers that are Pauli matrices:

$$\textsc{Pauli} \triangleq \{\mathbb{I}_2, X, Y, Z\} \tag{2.14}$$

$$\text{where } \mathbb{I}_2 \triangleq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, X \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Y \triangleq \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z \triangleq \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.15}$$

The n-qubit Pauli group is defined as the group generated by all *Pauli strings* $P_1 \otimes ... \otimes P_n$ with $P_i \in \textsc{Pauli}$, i.e., $\textsc{Pauli}_n \triangleq \langle \textsc{Pauli}^{\otimes n} \rangle$. One can check that $\textsc{Pauli}_n = \{i^c P_1 \otimes ... \otimes P_n \mid P_1, ..., P_n \in \textsc{Pauli}, c \in \{0,1,2,3\}\}$. In particular, we have $\textsc{Pauli}_1 = \{\pm P, \pm iP \mid P \in \textsc{Pauli}\}$ (the Pauli set with a factor ± 1 or $\pm i$). Pauli operators A, B either commute ($A \cdot B = B \cdot A$) or anticommute ($A \cdot B = -B \cdot A$).

Definition 2.1 (Stabilizer state, stabilizer group)**.** For a given n-qubit state $|\varphi\rangle$, its *stabilizer group* is the group of Pauli operators $P \in \textsc{Pauli}_n$ that stabilize $|\varphi\rangle$. This group is denoted $\text{Stab}(|\varphi\rangle)$. An element $P \in \text{Stab}(|\varphi\rangle)$ is called a *stabilizer* of $|\varphi\rangle$. A state $|\varphi\rangle$ is called a *stabilizer state* if it has 2^n stabilizers, i.e., if $|\text{Stab}(|\varphi\rangle)| = 2^n$. ◇

The stabilizer group of an n-qubit state always contains exactly 2^k elements for some $0 \leq k \leq n$. For any stabilizer group $G = \text{Stab}(|\varphi\rangle)$ with 2^k elements, there is a generating set S with $|S| = k$ elements. Moreover, no two stabilizer states have the same stabilizer group. Since (i) a stabilizer state's stabilizer group has n generators; (ii) each of which is a Pauli string of n Pauli matrices, plus a complex factor $c \in \{\pm 1, \pm i\}$ called the *phase*; and (iii) there are only 4 Pauli matrices, a stabilizer state can be uniquely described using only $\mathcal{O}(n^2)$ bits. It immediately follows that there are only $2^{\mathcal{O}(n^2)}$ n-qubit stabilizer states. More precisely, Aaronson and Gottesman (see [3], Proposition 2) show that the number of stabilizer states on n qubits grows asymptotically as $2^{(1/2+o(1))n^2}$.

Example 2.8. Examples of (generators of) stabilizer groups are $\text{Stab}(|0\rangle) = \langle Z\rangle$ and, letting $|\Phi_0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ be the Bell state, $\text{Stab}(|\Phi_0\rangle) = \langle X \otimes X, Z \otimes Z\rangle$.

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 the number of qubits [3, 136]. Various efficient algorithms exist for manipulating stabilizer (sub)groups G, including testing membership (is $A \in \textsc{Pauli}_n$ a member of G?) and finding a generating set of the intersection of two stabilizer (sub)groups. These algorithms predominantly use standard linear algebra, e.g., Gauss-Jordan elimination, as described in Sec. 3.5.2 in detail.

Example 2.9. We show how to use the stabilizer formalism to simulate the circuit in Figure 2.2. First, we use the explicit state vector representation; then we show the same simulation but using the stabilizer formalism. Here q_j denotes the j-th qubit register.

$$\begin{aligned}
|\varphi_0\rangle &= |000\rangle && \text{Initial state}\\
|\varphi_1\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|100\rangle && \text{applied } H \text{ to } q_1\\
|\varphi_2\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|110\rangle && \text{applied CNOT to control } q_1 \text{ and target } q_2\\
|\varphi_3\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|111\rangle && \text{applied CNOT to control } q_1 \text{ and target } q_3
\end{aligned}$$

To perform the simulation of this circuit using the stabilizer formalism, we identify each state $|\varphi_t\rangle$ with its stabilizer group $G(\varphi_t) = \{G_1, G_2, G_3\}$, as follows.

$$\begin{aligned}
G(\varphi_0) &= \{Z\mathbb{I}\mathbb{I}, \mathbb{I}Z\mathbb{I}, \mathbb{I}\mathbb{I}Z\} && \text{Initial state}\\
G(\varphi_1) &= \{\mathbb{I}\mathbb{I}Z, \mathbb{I}Z\mathbb{I}, X\mathbb{I}\mathbb{I}\} && \text{applied } H \text{ to } q_1\\
G(\varphi_2) &= \{\mathbb{I}\mathbb{I}Z, ZZ\mathbb{I}, XX\mathbb{I}\} && \text{applied CNOT to control } q_1 \text{ and target } q_2\\
G(\varphi_3) &= \{ZZ\mathbb{I}, Z\mathbb{I}Z, XXX\} && \text{applied CNOT to control } q_1 \text{ and target } q_3
\end{aligned}$$

For the purposes of this Introduction, we omit the detailed algorithms for obtaining the new stabilizer group after applying a Clifford gate; the interested reader may consult [3].

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 controlled Z gates. Graph states form a strict subset of all stabilizer states that is also important in error correction and measurement-based quantum computing [153].

We remark that, in contrast to other work, as noted in Definition 2.1, we also consider the stabilizer groups of states that are not stabilizer states. In general, we will refer to any abelian subgroup of $\textsc{Pauli}_n$, not containing $-\mathbb{I}_2^{\otimes n}$, as an n-qubit *stabilizer subgroup*; such a group has $\leq n$ generators. Such objects are also studied in the context of simulating mixed states [24] and quantum error correction [135]. A state is uniquely defined by its stabilizer group G if and only if $|G| = 2^n$, i.e., the elements of the group G have a unique joint +1 eigenvector if and only if $|G| = 2^n$.

Example 2.10. Examples of stabilizer subgroups are $\mathrm{Stab}(\frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)) = \{\mathbb{I}_2\}$, $\mathrm{Stab}(|1\rangle) = \langle -Z\rangle$ and $\mathrm{Stab}(\frac{1}{\sqrt{3}}(|00\rangle + |11\rangle) + \frac{2}{\sqrt{3}}(|01\rangle + |10\rangle)) = \langle X \otimes X\rangle$.

2.3 Decision Diagrams

Decision Diagrams are the protagonists of the story told in this dissertation.

We start with the definition, below, of a Binary Decision Diagram (BDD). It will serve as a template for all decision diagrams that follow.

Definition 2.2 (Binary Decision Diagram (BDD)). A BDD is a rooted, directed acyclic graph. It has two leaves, labeled True (or 1) and False (or 0). A non-leaf node is called a *Shannon node*; it is labeled with (the index of) a variable and has two

Table 2.1: The quantum gates used in this thesis. We remark that all Pauli gates are, in particular, Clifford gates.

	Gate name	**Symbol**	**Matrix**	**num. of qubits**
Pauli gates	Identity	$\mathbb{I}$	$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	1
	Pauli X	X	$= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	1
	Pauli Y	Y	$= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	1
	Pauli Z	Z	$= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	1
Clifford gates	Hadamard	H	$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	1
	Phase shift	S	$= \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	1
	Controlled NOT	CX	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & X \end{bmatrix}$	2
	Controlled Y	CY	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & Y \end{bmatrix}$	2
	Controlled Z	CZ	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & Z \end{bmatrix}$	2
	Swap	Swap	$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	2
Other	T-gate	T	$= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$	1
	Toffoli	CCX	$= \begin{bmatrix} \mathbb{I} & 0 \\ 0 & CX \end{bmatrix}$	3

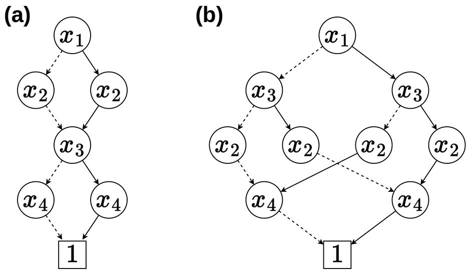


Figure 2.3: Two reduced, ordered BDDs representing the function $f(x_1, x_2, x_3, x_4) = (x_1 = x_2) \wedge (x_3 = x_4)$. BDD (a) uses the variable order $x_1 < x_2 < x_3 < x_4$; BDD (b) uses the variable order $x_1 < x_3 < x_2 < x_4$.

outgoing edges, called the *low edge* and the *high edge*. We will use the notation $-\!\textcircled{v}$ to denote an edge to node v, and $\textcircled{v_0}\cdots\cdots\textcircled{v}\longrightarrow\textcircled{v_1}$ to denote a node with a low edge to v_0 and a high edge to v_1. Each node v represents a Boolean function $[\![v]\!]$, defined inductively as follows. In the base case, the True and False leaves represent the constant functions $[\![\text{True}]\!] = 1$ and $[\![\text{False}]\!] = 0$, respectively. If a Shannon node v is labeled with variable x and has low edge to v_0 and high edge to v_1, then it represents the function $[\![v]\!] \triangleq \neg x \wedge [\![v_0]\!] \vee x \wedge [\![v_1]\!]$. A BDD is *ordered* if, on each path from the root to a leaf, each variable appears at most once and always in the same order. Two nodes u, v are called *equivalent* if they represent the same function, $[\![u]\!] = [\![v]\!]$. A BDD is *reduced* if there are no equivalent nodes. *Layer i* in an ordered BDD, is the set of nodes with variable label x_i. ◇

Example 2.11. Figures 2.3(a) and (b) show examples of BDDs. The two BDDs represent the same function, $f \triangleq (x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$. They have different shapes because they employ different variables orders, $x_1 < x_2 < x_3 < x_4$ and $x_1 < x_3 < x_2 < x_4$, respectively.

In the figure, the value $f(x)$ of an assignment x can be found by traversing the diagram from root to leaf as follows. One starts at the root node. A node is labeled with a variable x_i; if $x_i = 0$, we traverse the low (dotted) edge; otherwise, if $x_i = 1$, we traverse the high (solid) edge, until we arrive at a leaf, at which point we have found $f(x)$. More generally, a path from the root to a node v on layer k corresponds to a *partial assignment* $x_n = a_n, \ldots, x_{k-1} = a_{k-1}$ for some string $a \in \{0, 1\}^{n-k}$. This node represents the induced subfunction f_a. To avoid cluttering the diagram, edges to the False Leaf are not drawn in the figure.

To emphasize that the size of the BDD is influenced by the variable order, let us consider what happens if we generalize the function f and the variable orders from $n = 4$ to $n > 4$ variables. The function then becomes $f = (x_1 \Leftrightarrow x_2) \wedge \cdots \wedge (x_{n-1} \Leftrightarrow x_n)$ and the variable orders become $\sigma = x_1 < \cdots < x_n$ and $\pi = x_1 < x_3 < x_5 < \cdots < x_{n-1} < x_2 < x_4 \cdots < x_n$. Then the BDD with variable order σ has linear size, whereas the BDD with variable order π has exponential size (in the number of variables). In general, finding the optimal variable order for a given function is an NP-hard problem. [56].

Bryant [67] observed that BDDs can be queried and manipulated in polynomial time in the size of the diagrams (size is defined as the number of nodes). For example, (i) given a BDD of size m representing the function f, the number of models of f can be counted in time $\mathcal{O}(m)$; (ii) given BDDs f and g, with k and m nodes, respectively, a BDD representing the function $f \wedge g$ can be constructed in $\mathcal{O}(km)$ time.

A reduced and ordered BDD (ROBDD) is a canonical representation of its corresponding Boolean function [67]: a given function has a unique ROBDD. Canonicity ensures that the BDD is always as small as possible as equivalent nodes are merged. But more importantly, canonicity allows for quick equality checks: two reduced diagrams represent the same state *if and only if* the diagrams are exactly identical. In a practical decision diagram software package, this equality check can be performed in constant time: it suffices to check whether the two diagrams have the same root node, since the diagrams will share equivalent nodes in such a software package. More generally, canonicity allows for efficient manipulation algorithms through dynamic programming, which avoids traversing all paths (exponentially many in the size of the diagram in the worst case).

2.3.1 Decision diagrams for quantum states

The BDD, defined in Definition 2.2, can represent only Boolean functions, which are of the form $f\colon \{0,1\}^n \to \{0,1\}$. To represent quantum states, we use decision diagrams which are capable of representing *pseudo-Boolean* functions, of the form $f\colon \{0,1\}^n \to \mathbb{C}$. We review two prominent examples of decision diagrams that have this capability: the Quantum Information Decision Diagram (QuIDD, Definition 2.3) and the QMDD (Definition 2.4). The QuIDD extends the BDD by allowing more than two leaves: it contains one leaf for every element in the image of f. The interpretation of the QuIDD is similar to the BDD: to obtain the value $f(x)$ given a string $x \in \{0,1\}^n$, one simply traverses the diagram from root to leaf, traversing the low edge of a node when $x_j = 0$ and otherwise the high edge, when $x_j = 1$. The QuIDD is a specialization of an ADD; specifically, a QuIDD is an ADD in which the range of the function is restricted to the complex numbers. The QMDD (Definition 2.4) improves on the QuIDD (Definition 2.3) by merging nodes which are equivalent up to a complex factor. This complex factor is then stored as a label on the edges incident to the node that was merged. Because the QMDD merges nodes, it sometimes contains fewer nodes than the QuIDD. In the best case, the QMDD contains exponentially fewer nodes than the QuIDD. Given a QMDD representing a quantum state, measurement probabilities can be computed in polynomial time in the size of the QMDD. For all quantum gates there are algorithms to update the QMDD accordingly, i.e., given a gate U and a QMDD representing $|\varphi\rangle$, to construct a QMDD representing the state $U|\varphi\rangle$.

Definition 2.3 (Quantum Information Decision Diagram (QuIDD) [333])**.** A QuIDD is a rooted directed acyclic graph (DAG), representing a quantum state $|\varphi\rangle$. It has a leaf node for each unique amplitude of $|\varphi\rangle$, i.e., for each element in the set $\{\langle x|\varphi\rangle \mid x \in \{0,1\}^n\}$. A leaf labeled $z \in \mathbb{C}$ represents the length-1 vector $[\,z\,]$. Just as in a BDD, a non-leaf node is called a Shannon node. A Shannon node v with a low edge to node v_0, and a high edge to v_1, represents the quantum state $|v\rangle = |0\rangle \otimes |v_0\rangle + |1\rangle \otimes |v_1\rangle$. ◇

Definition 2.4 (Quantum Multi-valued Decision Diagram (QMDD) [227, 374])**.** A QMDD (QMDD) is a QuIDD in which (i) each edge is labeled with a complex value; and (ii) there is a unique leaf node, labeled with 1. An edge to node v with label z is denoted $\xrightarrow{z}\textcircled{v}$, and denotes the state $z|v\rangle$. A Shannon node with outgoing low edge $\xrightarrow{a}\textcircled{v_0}$ and high edge $\xrightarrow{b}\textcircled{v_1}$ is denoted $\textcircled{v_0}\overset{a}{\cdots\cdots}\textcircled{v}\overset{b}{\longrightarrow}\textcircled{v_1}$. Such a node represents the quantum state $|v\rangle = a|0\rangle \otimes |v_0\rangle + b|1\rangle|v_1\rangle$ (this is the Shannon decomposition from Equation 2.8). ◇

The DDs above all represent vectors. It is also possible to represent matrices using DDs [227, 333], but we defer this topic to Chapter 3.

Example 2.12. Figure 2.4 shows an example of a QMDD (d) and its construction from a binary tree (a).

It is instructive to consider the similarities and differences between the diagrams. In each type of decision diagram, a node v has two outgoing edges to nodes v_0, v_1, which represent functions on one fewer variable; the decision diagrams are all *ordered*, and

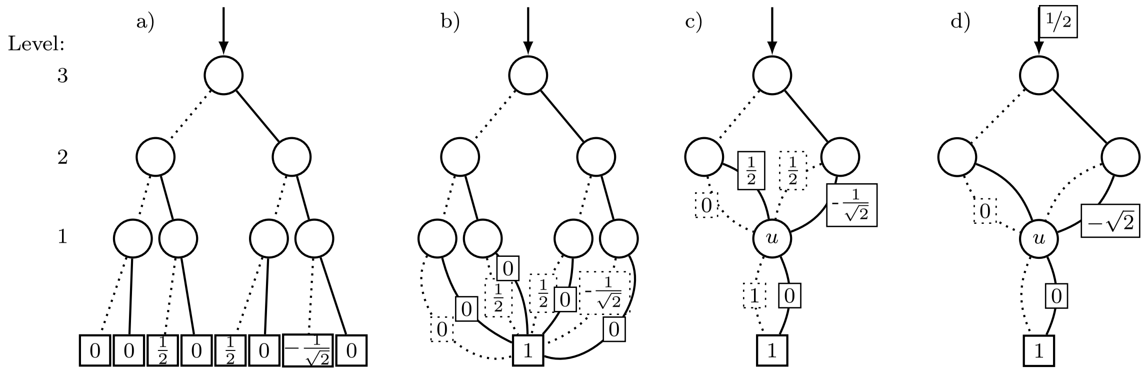


Figure 2.4: Different decision diagrams representing the 3-qubit state $[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0]^\top$, evolving into a QMDD (right). Left, (a) shows the exponential binary tree, where a node on level i represents x_i (see Equation 2.8) and its outgoing arrows $x_i = 0$ (dashed) and $x_i = 1$ (solid). The leaf contains the complex amplitude for the assignment corresponding to the path from the root. Next (b), the leafs are merged by dividing out common factors, putting these as weights (shown in boxes) on the edges of level 1 nodes (we can suppress a separate 0 leaf, as $0 = 0 \cdot 1$). Then the same trick is applied to level 1 nodes in (c). Here all level 1 nodes become *isomorphic* and can be merged into a new node u (note again that $0 \cdot |u\rangle = [0, 0]^\top$, where $|u\rangle = [1, 0]$ is the vector that node u represents). Finally, (d) shows the resulting QMDD, applying the same tactic to nodes on levels 2 and 3. Note that a QMDD requires a root edge. Merging (*isomorphic*) nodes makes QMDDs succinct. Adapted from Fig 2 in [374].

they can all be *reduced* by merging equivalent nodes. On the other hand, the various DDs differ in that they may label the edges with additional information. An edge e, labeled with a label ℓ, pointing to a node v, represents the state $|e\rangle = \ell \circ |v\rangle$, where $\circ$ is some appropriate notion of multiplication. Moreover, although all decision diagrams merge equivalent nodes, they understand equivalence differently. For example, the QuIDD considers two nodes u, v equivalent when they represent the same function, i.e., when $|u\rangle = |v\rangle$, whereas the QMDD considers two nodes equivalent when there exists a complex constant λ such that $\lambda |u\rangle = |v\rangle$. Lastly, different decision diagrams may impose different *reduction rules* in order to ensure canonicity.

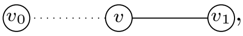
The field of decision diagrams is a fascinating and diverse field of research. Many decision diagrams have been proposed and implemented, of which Table 2.2 lists a small selection. The interested reader is directed to [344] for a more comprehensive treatment.

Lastly, although the definitions above are useful and unambiguous, it is often helpful to view decision diagrams from several complementary perspectives. We list some below.[§] We have found these different perspectives especially useful while designing new DDs and when trying to understand differences between DDs.

- A DD is a method of lossless data compression. A striking feature about decision diagrams is that they allow us to operate on the data *without decompressing it first.*

- A DD is a rooted DAG in which the set of paths from the root to the leaves correspond precisely to the assignments of f.

- A DD is a graphical depiction of the subfunctions of a function f. More precisely, the nodes of the diagram are in one-to-one correspondence with the induced subfunctions of f; and two subfunctions g, h of f are connected by an edge $g \to h$ iff h is a subfunction of g.

- A DD is a way to inductively define quantum states by using the Shannon decomposition. Namely, to define an amplitude vector $|\varphi\rangle$, whose top half is $|\varphi_0\rangle$, and whose bottom half is $|\varphi_1\rangle$, we first construct the DDs v_0, v_1 representing these two smaller vectors; then the vector $|\varphi\rangle$ can be represented by a node , whose two edges go to v_0 and v_1.

[§] We do not claim that all these perspectives are novel, only that, to the best of our knowledge, no source collects them into a list.

- A DD is obtained from a binary decision tree by merging equivalent nodes (under some suitable notion of equivalence).
- A DD is a finite state machine which computes a function $f(x)$ by reading an input string x and terminating after exactly $\mathrm{len}(x)$ steps. Possibly the machine is equipped with a small amount of internal memory to process the DD's edge labels.
- A DD is a Boolean (or algebraic, depending on the range) circuit composed of AND, OR and NOT gates (or multiplication and addition gates), with a single output gate. The output gate computes the function we are interested in. Not all circuits are decision diagrams, but every decision diagram is a circuit.

2.3.2 Comparing decision diagrams

The decision diagrams described above have different behaviour on the same problem, owing to their different reduction rules and merging strategies. Consequently, some decision diagrams may consume less memory, or less computation time, than another, for some given task. To quantify these differences, decision diagrams, and data structures more broadly, can be analytically compared on three criteria: succinctness, tractability, and rapidity. Succinctness is a partial order which tells us whether one data structure (DS) consumes less computer memory than another. Tractability tells us whether or not a given DS performs a given operation (such as applying a certain gate or measurement) in polynomial time in the size of the input diagram(s). Lastly, rapidity is a partial order which tells us whether one DS performs a given operation faster than another DS. Rapidity was introduced by Lai et al. [194]. Formal definitions follow.

Definition 2.5 (Succinctness, adapated from Darwiche and Marquis [97])**.** Let D_1, D_2 be two data structures. Then D_1 is *at least as succinct as* D_2, denoted $D_1 \preceq_s D_2$, iff there exists a polynomial p such that for every instance $\alpha \in D_2$ there exists an equivalent instance $\beta \in D_1$ such that $|\beta| \leq p(|\alpha|)$. Here $|\alpha|$ and $|\beta|$ are the sizes of α and β, respectively.

In this definition, the size $|\alpha|$ of a decision diagram is commonly taken to be the number of nodes in the diagram, or on the widest layer.

Table 2.2: Various decision diagrams (DDs) treated by the literature. A DD represents a function $f\colon \{0,1\}^n \rightarrow R$ where the column Range specifies the set R. Here S is an arbitrary algebraic structure. The column *Merging strategy* lists the conditions under which two nodes v, w, representing subfunctions $f, g\colon \{0,1\}^k \rightarrow \mathbb{R}$ are merged. Here $z, a \in \mathbb{C}$ are complex constants, P_i are Pauli gates and $f + a$ denotes the function defined by $f(x) + a$ for all x. The DDs in **bold underlined text** are treated in depth in this thesis. See Chapter 7 for a definition of *variable tree*; see Gergov and Meinel [129] for a definition of FBDD type.

Architecture	**(Quantum) decision diagrams (and variants)**	Range	**Node merging strategy**
variable order	Decision Tree	(any)	(no merging)
	BDD, [67] ZDD, [229] TBDD, [328] CBDD, [68] KFBDD, [103] DSDBDD, [46, 264] CCDD, [195] Partitioned ROBDD, [237] Mod-2-OBDD, [130] ROBDD[$\wedge_i$]$_{\mathcal{C}}$, [194] CFLOBDD [293]	$\{0,1\}$	$f = g$
	MTBDD [87], **QuIDD** [331]	$\mathbb{C}$	$f = g$
	ADD [27]	S	$f = g$
	SLDD$_\times$ [114, 352], **QMDD** [227, 374], TDD, [163] WCFLOBDD [296]	$\mathbb{C}$	$f = z \cdot g$
	SLDD$_+$ [114], EVBDD [193]	$\mathbb{C}$	$f = g + a$
	AADD [281], FEVBDD [308]	$\mathbb{C}$	$f = z \cdot g + a$
	LIMDD [337]	$\mathbb{C}$	$f = zP_1 \otimes ... \otimes P_n \cdot g$
	DDMF	$U(2)$	$M = \mathbb{I} \otimes \cdots \otimes id \otimes U \cdot R$
variable tree	**SDD**, [96] ZSDD, [245] TSDD, [111] VS-SDD [235]	$\{0,1\}$	$f = g$
	PSDD [180]	$\mathbb{R}$	$f = g$
FBDD type	FBDD [129], Partitioned ROBDD [237]	$\{0,1\}$	$f = g$

Definition 2.6 (Tractability, inferred from Darwiche and Marquis [97])**.** A data structure D *supports* a query or transformation OP iff there is a polynomial-time algorithm performing OP, taking as input (an) instance(s) of D. That is, the algorithm is polynomial in the size $|\alpha|$ of the instance α. In that case, the operation is said to be *tractable* for D.

Here, a *query* is a function mapping (tuples of) quantum states to some fixed range, whereas a *transformation* maps (tuples of) quantum states to a quantum state represented by the same data structure D. For example, computing the probability of a measurement outcome is a *query*; therefore, algorithms on both QuIDD and QMDD have the same range, namely, they output a real number in $[0, 1]$. By contrast, the addition of two quantum states is a *transformation*. Here, on input $(|\varphi\rangle, |\psi\rangle)$, the QuIDD (QMDD) algorithm for addition is expected to output a QuIDD (resp. QMDD) representing the vector $|\varphi\rangle + |\psi\rangle$. Therefore, the codomain of the addition algorithms of QuIDD and QMDD are different.

Definition 2.7 (Rapidity, adapted from Lai et al. [194])**.** Let D_1, D_2 be two canonical data structures and OP an operation. Then D_1 performs OP *at least as rapidly* as D_2 iff, for each algorithm ALG_1 performing $OP(D_2)$, there exists some polynomial p and an algorithm ALG_1 performing $OP(D_1)$ such that, for every input x to $OP(D_2)$, and its equivalent input y to $OP(D_1)$, the runtime of ALG_1 is $p(t_2 + |x|)$, where t_2 is the runtime of ALG_2 on y.

We emphasize that this definition requires both data structures to be canonical. For non-canonical data structures, "*the equivalent input*" is not defined, as there may be multiple instances of D_1 representing the same quantum state, possibly having different sizes. In Chapter 5, we give a generalized definition which drops the requirement that the data structures are canonical.

2.4 Approaches to quantum software verification

In this section, we list several approaches to quantum software verification. This establishes the context of the work of this thesis, but is not intended as an exhaustive survey of the topic. For a more in-depth treatment, we refer the reader to Ying and Feng's book on quantum model checking [361], their survey [360] and to an accessible introduction to the topic by Turrini [317].

We divide these contributions into *specification languages*, and *software tools*. A specification language for quantum programs is a logic in which a user can specify the permitted behaviour or output of a quantum program, including how the (quantum) state of the program may evolve over time (very broadly speaking). A software tool puts such a specification language into practice by providing a practical method to check whether this quantum protocol satisfies its specification. Several existing tools provide support for quantum programming languages: they allow programmers to write their quantum programs in a high-level programming language, as opposed to defining a quantum circuit gate by gate. Many of these languages support abstract data types, such as integers, Boolean values and qubits. Assertions can then be formulated inside the code, e.g., as preconditions and postconditions, which the model checking tool can then verify. Several of the tools mentioned below build on two established quantum programming languages: QPL [285] and Q# [306]. We refer the interested reader to the surveys on quantum programming languages of Garwahl et al. [126] and Ferreina [118].

Relation to the present work. Relative to the work below, in this work we adopt the simplest "specification language," namely, we aim to simply check whether two circuits are equivalent. We present a software tool which lays the groundwork for this purpose in Chapter 3 and Chapter 4. We discuss possibilities for extending the present work to more expressive specification languages and logics, such as those described below, in Section 8.3.

Specification languages. A specification language is a logic which specifies how a state may evolve over time. Most such logics can be thought of, informally, as being constructed in two steps: first, we define a set of atomic propositions, which allows us to express assertions about quantum states; in the second step, we add temporal operators, which allows us to express assertions about how quantum states evolve over the course of the program, e.g., which quantum states are supposed to be reachable in the program. The second step makes the logic a quantum *temporal* logic. By analogy, in the classical domain, CTL can be obtained using this recipe by starting with propositional logic, which we can use to formulate assertions about states of a system, and then adding temporal operators. Similarly, quantum CTL [30] (see below) can be obtained by starting with so-called quantum propositional logic (EQPL [217]) and adding temporal operators. The approaches listed below use one of three methods for expressing assertions about quantum states, thus accomplishing the first step in our recipe: they use either (i) exogenous quantum propositional logic (EQPL), defined by Mateus and Sernadas [217]; or (ii) quantum preconditions and postconditions, formulated by D'Hondt and Panangaden [100]; or (iii) closed subspaces of the Hilbert space, used by Ying in [363]. Below, we start by treating work based on the EQPL formalism of Mateus and Sernadas, followed by work which builds on D'Hondt and Panangaden's approach, and lastly we mention Ying's quantum LTL based on closed subspaces.

Mateus and Sernadas introduce *exogenous quantum propositional logic* (EQPL), which allows one to express assertions about pure quantum states, e.g., that a given subset of the qubits is not entangled with another subset [217]. Chadha, Mateus and Sernadas extend this logic to mixed states (i.e., probability distributions over quantum states), obtaining *Ensemble EQPL* (EEQPL) [79].

Baltazar, Chadha and Mateus introduce quantum computation tree logic (QCTL) [30]. This logic is obtained from the EQPL of Mateus and Sernadas [217] by adding the usual temporal operators of CTL. For example, one can specify that *eventually*, a given subset of qubits will be entangled with another subset.

Chadha, Mateus and Sernadas formulate a Hoare-style logic for quantum programs, in which preconditions and postconditions are assertions in EEQPL [79]. They formulate axioms and inference rules for this logic and show that the resulting calculus is sound. Their Hoare logic can reason only about so-called quantum while-programs with bounded iteration.

The literature contains two approaches to formulating a quantum analogue of linear time logic, by Mateus et al. [216] and by Ying, Li and Feng [363]. First, Mateus et al. [216] introduce a quantum linear time logic (QLTL) based on the EQPL of Mateus and Sernadas [217]. In QLTL, one can assert, for example, that in the next computation step, some subset of qubits is unentangled with another set of qubits. Building on the QCTL of Baltazar et al. [30], Mateus et al. study the satisfiability and model checking problems of QCTL and QLTL.

Second, Ying, Li and Feng [363] introduce another quantum analogue of linear-time logic. In this logic, an atomic proposition asserts that a state $|\psi\rangle$ is an element of a set X, where X is a closed subspace of the Hilbert space. A linear-time property P is then defined as a (finite or infinite) collection of (finite or infinite) sequences of conjunctions of atomic propositions. They use the quantum automaton [186] as their model of computation. A (finite or infinite) sequence of states of this automaton is called a *trace*, just as in the classical sense. Let $A = A_0, A_1, \ldots \in P$ be an element of a linear-time property P, i.e., each A_j is a conjunction of atomic propositions. Then a trace $\pi = |\varphi_0\rangle, |\varphi_1\rangle, \ldots$ of an automaton satisfies A if $|\varphi_t\rangle \in X$ for each $X \in A_t$ at each time $t \geq 0$. An automaton satisfies a given predicate P if all the automaton's possible traces satisfy some $A \in P$. Ying, Li and Feng provide an algorithm for checking whether an automaton satisfies a given invariant.

D'Hondt and Panangaden formulate a notion of pre- and postconditions which takes a slightly different approach than the propositional logic approach above [100]. Their assertions are Hermitian operators P satisfying $0 \leq P \leq \mathbb{I}$.[¶] For a given state $|\varphi\rangle$ and an assertion P (i.e., a Hermitian operator), D'Hondt and Panangaden argue that the value $0 \leq \langle\varphi| P |\varphi\rangle \leq 1$ should be interpreted, informally speaking, as the probability that $|\varphi\rangle$ satisfies P. This is analogous to a classical probabilistic statement: if μ is a probability distribution and P is a classical predicate, then the state satisfies P with probability $0 \leq \mu.P \leq 1$. With this in mind, D'Hondt and Panangaden go on to define that, for a given quantum gate U and Hermitian operators P, Q, the triple $\{P\}U\{Q\}$ constitutes a valid Hoare triple if, for all states $|\varphi\rangle$, it holds that $P \leq U^\dagger Q U$. Consequently, if $\{P\}U\{Q\}$ is a valid Hoare triple, then a quantum state will satisfy Q with probability p after application of the gate U if it satisfied P with probability p before applying the gate. This formalization naturally lends itself to mixed states in addition to pure states.

Ying uses the Hoare triples of D'Hondt and Panangaden to establish a full-fledged quantum Hoare logic for quantum while-programs [359]. He gives deduction rules for this logic and formulates and proves soundness and completeness for partial and total correctness. Relative to the quantum Hoare logic of Chadha et al. [79], the quantum Hoare logic of Ying is based on D'Hondt and Panangaden's approach in which an assertion is a Hermitian operator, rather than a statement in EQPL [217], as in Chadha et al.'s case. Zhou et al. consider the special case of this Hoare logic in which the assertions are restricted to be projections, rather than general Hermitian operators [367]. They argue that this is a common use case and suffices to prove correctness of several quantum programs. Since such assertions may be easier to generate and check, this justifies the penalty to expressiveness.

Software tools. We first list tools that model check specifications given in a temporal logic, after which we list tools that check equivalence of two quantum circuits or programs.

[¶] Here $A \leq B$ means $\langle\varphi| A |\varphi\rangle \leq \langle\varphi| B |\varphi\rangle$ for all quantum states $|\varphi\rangle$. Therefore, requiring that $0 \leq A \leq \mathbb{I}$ ensures that $0 \leq \langle\varphi| A |\varphi\rangle \leq 1$ for all states $|\varphi\rangle$.

First, Gay and Nagarajan [128] build a model checker which can check QCTL properties of a given quantum protocol. The quantum protocols that are considered are restricted: the only quantum gates that are allowed are the Clifford gates. Since the Clifford gate set is not universal for quantum computing, not all quantum protocols can be expressed in this tool.

Honarvar and Nagarajan [161] implement a model checker for programs written in the Q# quantum programming language [306]. They allow a user to formulate a precondition and postcondition for any program statement, inspired by the quantum Hoare logics of Ying et al. [359] and Zhou et al. [367] above.

Feng et al. [116] build QPMC, which can check QCTL properties (see [30]) of quantum programs. To this end, they develop a quantum programming language by extending the PRISM guarded-command based language [191]. QPMC extends the probabilistic model checker IScasMC [147] and, contrary to the model checker of Gay and Nagarajan [128], can check general quantum programs, as it does not restrict the gate set. Internally, QPMC models a program as a quantum Markov chain, a notion introduced by Feng et al. [117].

Liu et al. [205] formalize the quantum Hoare logic of Ying [359] in the proof assistant Isabelle/HOL [244]. They use this formalization to prove the correctness of an implementation of Grover's algorithm.

Ardeshir-Larijani, Gay and Nagarajan build two equivalence checking tools: first, an equivalence checker for quantum programs and protocols [17] that are written in the *Quantum Programming Language* (QPL, introduced by Selinger [285]); and second, an equivalence checker for concurrent quantum programs [18]. In [18], Ardeshir-Larijani et al. develop a new quantum programming language for concurrent quantum processes. This programming language is inspired by qCCS of Ying et al., an algebra of concurrent quantum processes [362]. Just as in [128], the quantum protcols that are considered restricted to use only Clifford gates.

Burgholzer et al. provide two tools for quantum circuit equivalence checking. First, given circuits $U = U_1 \cdots U_m$ and $V = V_1 \dots V_\ell$, Burgholer and Wille [75] use QMDDs to construct the unitary matrix $U^\dagger V$. If the circuits are equivalent, i.e., if $U = \lambda V$, then $U^\dagger V = \lambda \mathbb{I}$, which is a matrix with a very simple QMDD. With this in mind, they construct the matrix $U^\dagger V$ by iteratively constructing matrices of the form $(U_1 \cdots U_a)^\dagger \cdot (V_1 \cdots V_b)$ for some $a \leq m, b \leq \ell$, finding empirically that these intermediate matrices often remain close to the identity and hence, have small QMDDs.

Second, Burgholzer, Kueng and Wille [71] use `QMDD`s to simulate two given circuits on a randomly chosen stabilizer state. If the circuits are found to yield different output states, they conclude the circuits are not equivalent. Although this method is sound but not complete, they show that the probability of finding a counterexample is quite good if U and V are "very different," in a precise sense. Specifically, they consider the *average fidelity* $\mathcal{F}_{\text{avg}}(U, V)$, of two unitary matrices, defined below. They show that for a random stabilizer state $|g\rangle$, and unitaries U, V, it holds that

$$\mathbb{E}_{|g\rangle}[\langle g| U^\dagger V |g\rangle] \approx \mathcal{F}_{\text{avg}}(U, V) = \frac{1}{2^n + 1}\left(1 + 2^n \left|\text{tr}(U^\dagger V)\right|^2\right) \tag{2.16}$$

Using Markov's inequality, we immediately obtain a bound on the probability that the circuits yield the same result:

$$\mathbb{P}_{|g\rangle}[| \langle g| U^\dagger V |g\rangle |^2 = 1] \leq \mathcal{F}_{\text{avg}}(U, V) \tag{2.17}$$

Notably, the expectation value achieved by random stabilizer states in Equation 2.16 is the same as that achieved by a random Haar state. This is a consequence of the fact that the set of stabilizer states is a complex projective 3-design [190]. In this narrow sense, therefore, using random stabilizer states is *optimal* in the black box setting. However, random stabilizer states are not optimal in the sense that the probability of obtaining a counterexample is not as high as for a random Haar state. To achieve this, one would need to choose as random inputs a different set of states, which is a projective t-design for some large t.

Lastly, Hong et al. [162] implement a tool for quantum circuit equivalence checking in the presence of noise. This equivalence checking of noisy circuits is more general than the equivalence checking problem studied by Burgholzer et al. above [71, 75]. The noisy circuit is modeled by a superoperator, rather than a unitary matrix. They represent the noisy circuit as a tensor network using the Tensor Decision Diagram (TDD); they then contract the tensor network using operations on the TDD.

Chapter 3

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

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

This chapter contributes to answering Research Question 1:

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

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

3.1 Introduction

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

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

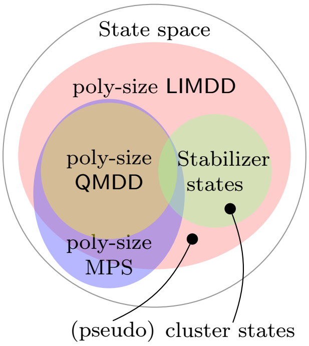


Figure 3.1: The set of stabilizer states and states representable as poly-sized: (Pauli-)LIMDDs (this work), QMDDs and MPS.

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

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

Further, we give algorithms for simulating quantum circuits using Pauli-LIMDDs. We continue by investigating the power of these algorithms compared to state-of-the-art simulation algorithms based on QMDD, MPS and stabilizer decomposition. We find circuit families which Pauli-LIMDD can efficiently simulate, which stands in stark contrast to the exponential space needed by QMDD-based, MPS-based and a stabilizer-decomposition-based simulator (the latter result is conditional on the exponential time hypothesis). This is the first analytical comparison between decision diagrams and matrix product states.

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

The canonicity algorithm effectively determines whether two n-qubit quantum states $|\varphi\rangle, |\psi\rangle$, each represented by a LIMDD node φ, ψ, are equivalent up to a Pauli operator P, i.e, $|\varphi\rangle = P|\psi\rangle$, which we call an isomorphism between $|\varphi\rangle$ and $|\psi\rangle$. Here $P = P_n \otimes ... \otimes P_1$ consists of single qubit Pauli operators P_i (ignoring scalars for now). In general, there are multiple choices for P, so the goal is to make a deterministic selection among them, to ensure canonicity of the resulting LIMDD. To do so, we first take out one qubit and write the states as, e.g., $|\varphi\rangle = c_0|0\rangle|\varphi_0\rangle + c_1|1\rangle|\varphi_1\rangle$ for complex coefficients c_0, c_1. We then realize that $P_{\text{rest}} = P_{n-1} ... \otimes P_1$ must map the pair $(|\varphi_0\rangle, |\varphi_1\rangle)$ to either $(|\psi_0\rangle, |\psi_1\rangle)$ or $(|\psi_1\rangle, |\psi_0\rangle)$ (in case P_n is a diagonal or antidiagonal, respectively). Hence P_{rest} is a member of the intersection of the two sets of isomorphisms. Next, we realize that the set of all isomorphisms, e.g. mapping $|\varphi_0\rangle$ to $|\psi_0\rangle$, is a coset $\pi \cdot G$ of the stabilizer group G of $|\varphi_0\rangle$ (i.e. the set of isomorphisms mapping $|\varphi_0\rangle$ to itself) where π is a single isomorphism $|\varphi_0\rangle \rightarrow |\psi_0\rangle$. Thus, to find a (canonical) isomorphism between n-qubit states $|\varphi\rangle \rightarrow |\psi\rangle$ (or determine no such isomorphism exists), we need three algorithms: to find (a) an isomorphism between $(n-1)$-qubit states, (b) the stabilizer group of an $(n-1)$-qubit state (in fact: the group generators, which form an efficient description), (c) the intersection of two cosets in the Pauli group (solving the *Pauli coset intersection problem*). Task (a) and (b) are naturally formulated as recursive algorithms on the number of qubits, which invoke each other in the recursion step. For (c) we provide a separate algorithm which first rotates the two cosets such that one is a member of the Pauli Z group, hence isomorphic to a binary vector space, followed by using existing algorithms for binary coset (hyperplane) intersection. Having characterized all isomorphisms $|\varphi\rangle \to |\psi\rangle$, we select the lexicographical minimum to ensure canonicity. We emphasize that the algorithm works for arbitrary quantum states, not only stabilizer states.

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

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

3.2 Preliminaries: decision diagrams and the stabilizer formalism

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

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

3.2.1 Decision diagrams

An n-qubit quantum state $|\varphi\rangle$ can be represented as a 2^n-dimensional vector of complex numbers (modeling amplitudes) and can thus be described by a pseudo-Boolean function $f : \{0,1\}^n \to \mathbb{C}$ where

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

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

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

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

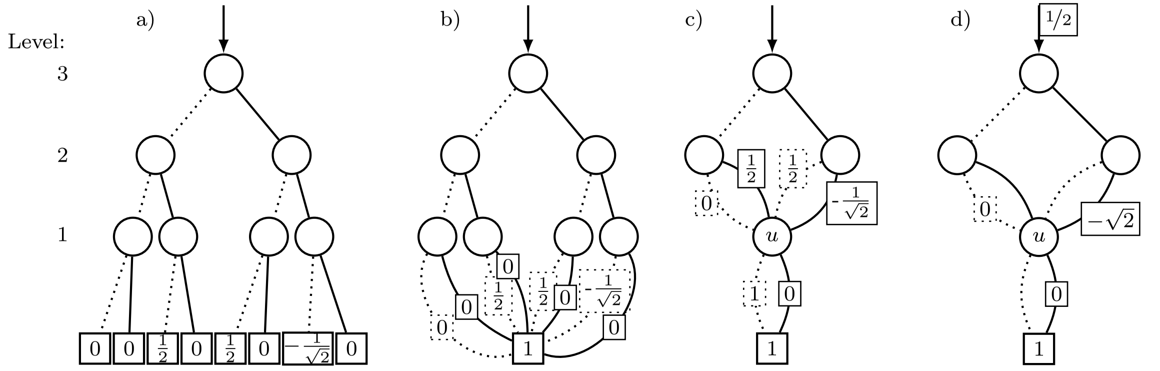


Figure 3.2: Different decision diagrams representing the 3-qubit state $[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0]^\top$, evolving into a QMDD (right). Left, (a) shows the exponential binary tree, where a node on level i represents x_i (see Equation 3.1) and its outgoing arrows $x_i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

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

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

3.2.2 Pauli operators and stabilizer states

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

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

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

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

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

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

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

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

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

In this work, we also consider states which are not stabilizer states and which therefore have a nonmaximal stabilizer group (i.e. $< n$ generators). To emphasize that a stabilizer group need not be maximal, i.e. it is a subgroup of maximal stabilizer groups, we will use the term *stabilizer subgroup*. Such objects are also studied in the context of simulating mixed states [24] and quantum error correction [135]. Examples of stabilizer subgroups are $\{\mathbb{I}\}$ for $\frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)$, $\langle -Z\rangle$ for $|1\rangle$ and $\langle X \otimes X\rangle$ for $\frac{1}{\sqrt{6}}(|00\rangle + |11\rangle) + \frac{1}{\sqrt{3}}(|01\rangle + |10\rangle)$. In contrast to stabilizer states, in general a state is not uniquely identified by its stabilizer subgroup.

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

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

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

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

3.2.3 Matrix product states

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

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

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

3.3 Local Invertible Map Decision Diagrams

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

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

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

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

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

There is a family of circuits C such that:

---

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

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

3.3.1 The LIMDD data structure

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

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

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

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

---

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

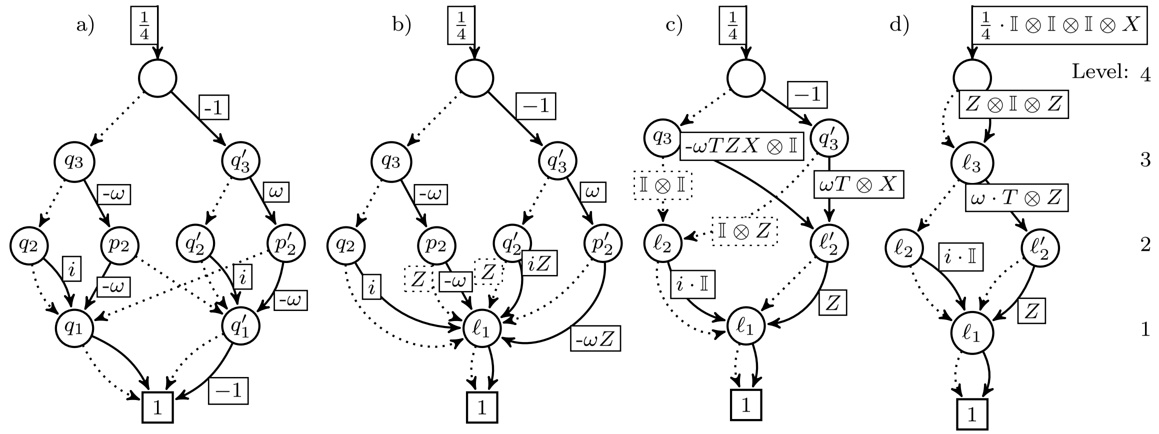


Figure 3.3: A QMDD (a) representing the state $\frac{1}{4}[1, 1, i, i, -\omega, \omega, i, i, -1, 1, -i, i, -\omega, -\omega, i, -i]^\top$ with $\omega = e^{i\pi/4}$, evolving into a LIMDD (d). As in Figure 3.2, diagram nodes are horizontally ordered in 'levels' with qubit indices $4, 3, 2, 1$. Low edges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

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

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

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

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

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

- $\mathsf{low}, \mathsf{high} \colon \textsc{Node} \to \textsc{Node} \cup \{\mathsf{Leaf}\}$ indicate the low and high edge functions, respectively. We write low_v (or high_v) to obtain the edge (v, w) with $w = \mathsf{low}(v)$ (or $w = \mathsf{high}(v)$). For all $v \in \textsc{Node}$ it holds that $\mathsf{idx}(\mathsf{low}(v)) = \mathsf{idx}(\mathsf{high}(v)) = \mathsf{idx}(v) - 1$ (no qubits are skipped[‡]);

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

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

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

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

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

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

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

---

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

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

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

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

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

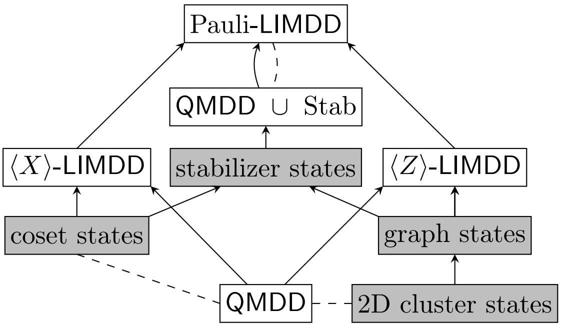


Figure 3.4: Relations between non-universal classes of quantum states (gray) and decision diagrams, where we consider a diagram as the set of states that it can represent in polynomial size. Solid arrows denote set inclusion. Dashed arrows $D_1 \dashrightarrow D_2$ signify an exponential separation between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

3.3.2 Succinctness of LIMDDs

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

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

Figure 3.4 visualizes succinctness relations between different quantum state representations, as proved in Prop. 3.2. In particular, G-LIMDDs with $G = \langle \textsc{Pauli} \rangle$ can be exponentially more succinct than QMDDs, and retain this exponential advantage even with $G = \langle Z \rangle, \langle X \rangle$. In Corollary 3.1, we show the strongest result, namely that LIMDDs are also more succinct than the union of QMDDs and stabilizer states, written QMDD ∪ Stab, which can be thought of a structure that switches between QMDD and the stabilizer formalism depending on its content (stabilizer or non-stabilizer state). This demonstrates that ad-hoc combinations of existing formalisms do not make LIMDDs obsolete.

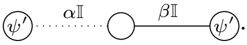
Proposition 3.2. The inclusions and separations in Figure 3.4 hold.

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

Theorem 3.1 shows that any stabilizer state can be represented as a ⟨PAULI⟩-Tower LIMDD (Definition 3.3).

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

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

Proof sketch of Theorem 3.1. (Full proof in App. B) The $n = 1$ case: the six single-qubit states $|0\rangle, |1\rangle, |0\rangle \pm |1\rangle$ and $|0\rangle \pm i\,|1\rangle$ are all represented by a ⟨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 

. 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

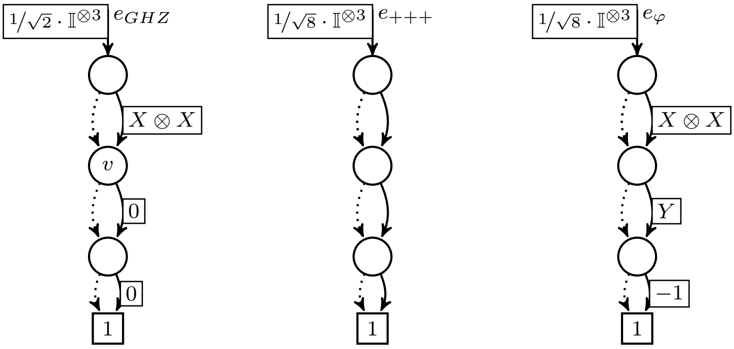
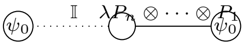
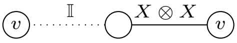

$1/\sqrt{2} \cdot \mathbb{I}^{\otimes 3}$
e_{GHZ}
$X \otimes X$
v
0
0
1
$1/\sqrt{8} \cdot \mathbb{I}^{\otimes 3}$
e_{+++}
1
$1/\sqrt{8} \cdot \mathbb{I}^{\otimes 3}$
e_{φ}
$X \otimes X$
Y
-1
1

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

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

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

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

We stress that obtaining the LIMs for the Pauli Tower-LIMDD of a stabilizer state is not immediate from the stabilizer generators; specifically, the edge labels in the Pauli-LIMDD are not directly the stabilizers of the state. For example, the *GHZ* state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$ is represented by $|e_{GHZ}\rangle = \frac{1}{\sqrt{2}}$  with $|v\rangle = |00\rangle$ in Figure 3.5, but $X \otimes X$ is not a stabilizer of $|00\rangle$. Nonetheless, Theorem 3.1 implicitly contains an algorithm that constructs a ⟨Pauli⟩-Tower LIMDD stabilizer state. Section 3.4 also provides the inverse construction, which we use to make LIMDDs (representing any quantum state) canonical in time $\mathcal{O}(mn^3)$ (using Algorithm 3).

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

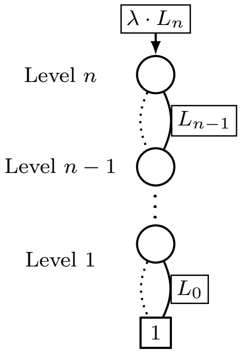


Figure 3.6: Figure 3.6: An n-qubit G-Tower LIMDD. We let $L_i \in G^{\otimes i} \cup \{0\}$ and $\lambda \in \mathbb{C} \setminus \{0\}$ (only root edges have an arbitrary scalar).

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

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

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

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

---

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

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

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

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

3.3.2.2 LIMDDs are exponentially more succinct than matrix product states

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

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

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

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

3.3.3 Pauli-LIMDD manipulation algorithms for simulation of quantum computing

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

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

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

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

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

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

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

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

```
procedure MakeEdge(Edge $\xrightarrow{A}\textcircled{v}$, Edge $\xrightarrow{B}\textcircled{w}$)
    $u := \textcircled{v}\overset{A}{\cdots\cdots}\textcircled{u}\xrightarrow{B}\textcircled{w}$
    return Edge $\xrightarrow{\mathbb{I}^{\otimes k}}\textcircled{u}$                ▷ Where $k = \mathsf{idx}(u)$
```

Algorithm 3 Make any LIMDD canonical using MakeEdge.

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

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

Finally, in this section, we often decompose LIMS using $A = \lambda P_n \otimes P'$. Here $\lambda \in \mathbb{C}$ is a non-zero scalar, P' a Pauli string and $P_n \in \{\mathbb{I}, X, Z, Y\} = \textsc{Pauli}$. Our algorithms will use the Follow procedure from Algorithm 4 to easily navigate diagrams according to edge semantics. Provided with a bit string $x_n ... x_1$, the procedure is the same as ReadAmplitude. If however fewer bits are supplied, it returns a LIMDD root edge representing a subvector. For instance, the subvector for $|10\rangle$ of the LIMDD root edge e_r in Figure 3.3 (d) is computed by taking $|\textsc{follow}(10, e_r)\rangle = |\xrightarrow{\frac{1}{4}\mathbb{I}\otimes XZ}\textcircled{ℓ_2}\rangle = \frac{1}{4}\cdot[-1, 1, -i, i]$. So, we can specify it as $|\textsc{follow}(b, e)\rangle = (\langle b| \otimes \mathbb{I}^{n-\ell})\,|e\rangle$, i.e., select the bth block of size $2^{n-\ell}$ from the vector $|e\rangle$ (or rather, return a LIMDD edge representing that block).

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

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

3.3.3.1 Performing a measurement in the computational basis

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

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

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

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

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

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

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

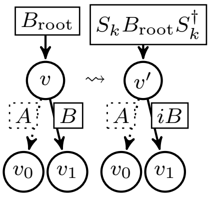
3.3.3.2 Gates with simple LIMDD algorithms

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

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

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

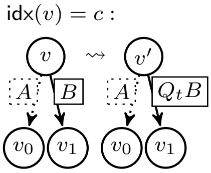
Applying any diagonal or antidiagonal single-qubit gate to the top qubit can be done efficiently, e.g., applying the T**-gate to the top qubit**. For root edge $e = \xrightarrow{B_{\text{root}}} \textcircled{v}$, we can construct $e_x = \textsc{follow}(x, e)$, which propagates the root edge's LIM to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



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

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

```
1: procedure SGATE(EDGE $\xrightarrow{A}\textcircled{v}$ with $A \in$ PAULI-LIM, $k \in \{1, ..., \mathsf{idx}(v)\}$)
2:     if $v \notin$ SGATECACHE then                          ▷ Compute result once for $v$ and store in cache:
3:         if $\mathsf{idx}(v) = k$ then
4:             SGATECACHE[$v$] := MAKEEDGE($\mathsf{low}_v, i \cdot \mathsf{high}_v$)
5:         else
6:             SGATECACHE[$v$] := MAKEEDGE(SGate($\mathsf{low}_v, k$), SGate($\mathsf{high}_v, k$))
7:     return $S_k A S_k^\dagger \cdot$ SGATECACHE[$v$]                ▷ Retrieve result from cache
```



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

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

```
1: procedure CPAULIGATE(EDGE $\xrightarrow{A}\textcircled{v}$ with $A \in$ PAULI-LIM, $c$, $t$ with $1 \leq c < t \leq n$)
2:     if $v \notin$ CPAULICACHE then                         ▷ Compute result once for $v$ and store in cache:
3:         if $\mathsf{idx}(v) = k$ then
4:             CPAULICACHE[$v$] := MAKEEDGE($\mathsf{low}_v, X_t \cdot \mathsf{high}_v$)
5:         else
6:             CPAULICACHE[$v$] := MAKEEDGE(CPauliGate($\mathsf{low}_v, c, t$), CPauliGate($\mathsf{high}_v, c, t$))
7:     return $CX_t^c \cdot A \cdot CX_t^{c\dagger} \cdot$ CPAULICACHE[$v$]        ▷ Retrieve result from cache
```

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

3.3.3.3 Applying a generic multi-qubit gate to a state

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

$$u = \overbrace{\begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}}^{u_{0*}} \left.\vphantom{\begin{bmatrix} u_{00} \\ u_{10} \end{bmatrix}}\right\} u_{*1} \tag{3.6}$$

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

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

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

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

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

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

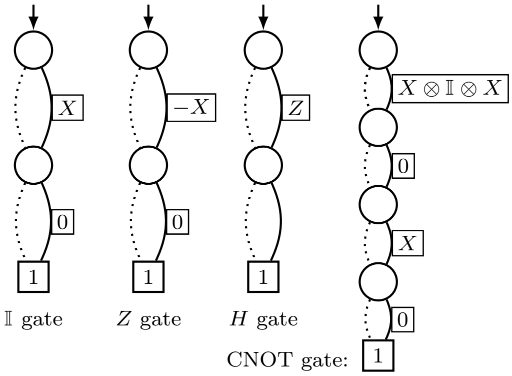


Figure 3.7: LIMDDs representing various gates.

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

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

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

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

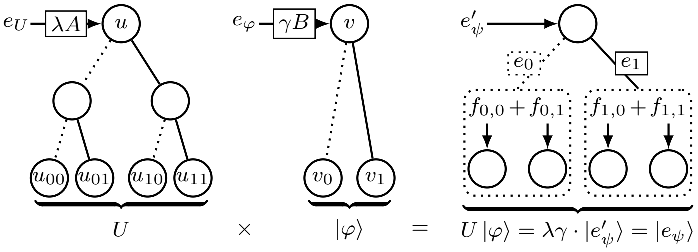


Figure 3.8: An illustration of ApplyGate (Algorithm 8), where matrix U is applied to state $B\,|v\rangle$, both represented as Pauli-LIMDDs. The edges $f_{0,0}$, $f_{0,1}$, etc. are the edges made on Line 6. The dotted box indicates that these states are added, using Add, producing edges e_0, e_1, which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

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

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

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

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

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

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

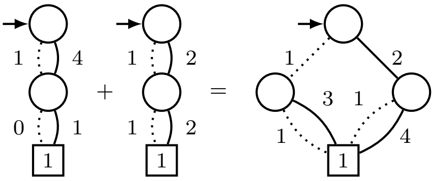


Figure 3.9: Adding two states $(0, 1, 0, 4)$ and $(1, 2, 2, 4)$ as QMDDs can cause an exponentially larger result QMDD $(1, 3, 2, 8)$ due to the loss of common factors.

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

The worst-case runtime of ADD is $\mathcal{O}(n^3 2^n)$ (exponential as expected), where n is the number of qubits. This can happen when the resulting LIMDD is exponential in the input sizes (bounded by 2^n), as identified for QMDDs in [113, Table 2]. The reason for this is that addition may remove any common factors, as illustrated in Figure 3.9. However, the ADD algorithm is polynomial-time when $v = w$ and v is a stabilizer state, which is sufficient to show that the Hadamard gate can be efficiently applied to stabilizers represented as LIMDD, as we demonstrate next in Sec. 3.3.3.4.

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

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

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

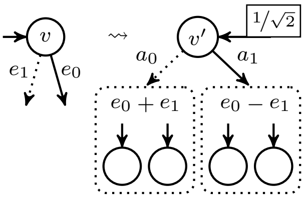
Theorem 3.4. Any Clifford gate (H, S, CNOT) can be applied in $\mathcal{O}(n^4)$ time to any (combination of) qubits to a LIMDD representing a stabilizer state.

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

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

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

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



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

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

```
1: procedure HGate(Edge $\xrightarrow{A}\textcircled{v}$ with $A \in$ Pauli-LIM, $k \in \{1,\ldots,\mathsf{idx}(v)\}$)
2:     if $v \notin$ HGateCache then            ▷ Compute result once for $v$ and store in cache:
3:         if $\mathsf{idx}(v) = k$ then
4:             $a_0 :=$ Add($\mathsf{low}(v), \mathsf{high}(v)$)
5:             $a_1 :=$ Add($\mathsf{low}(v), -\mathsf{high}(v)$)
6:             HGateCache[$v$] := $1/\sqrt{2} \cdot$ MakeEdge($a_0, a_1$)
7:         else
8:             HGateCache[$v$] := MakeEdge(HGate($\mathsf{low}(v), k$), HGate($\mathsf{high}(v), k$))
9:     return $H_k A H_k^\dagger \cdot$ HGateCache[$v$]            ▷ Retrieve result from cache
```

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

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

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

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

3.3.4 Comparing LIMDD-based simulation with other methods

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

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

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

3.3.4.1 LIMDD is exponentially faster than QMDD-based simulation

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

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

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

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

3.3.4.2 LIMDD is exponentially faster than MPS

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

3.3.4.3 LIMDD is exponentially faster than Clifford + T

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

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

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

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

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

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

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

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

Lastly, it would be interesting to analytically compare LIMDD with general stabilizer rank based simulation (without assuming ETH). However, this would require finding a family of states with provably superpolynomial stabilizer rank, which is a major open problem. Instead, we implemented a heuristic algorithm by Bravyi et al. [61] to empirically find upper bounds on the stabilizer rank and applied it to a superset of the W states, so-called Dicke states, which can be represented as polynomial-size LIMDD. The $\mathcal{O}(n^2)$-size LIMDD can be obtained via a construction by Bryant [67], since the amplitude function of a Dicke state is a symmetric function. The results hint at a possible separation but are inconclusive due to the small number of qubits which the algorithm can feasibly investigate in practice. See Section 3.6 for details.

3.4 Canonicity: Reduced LIMDDs with efficient MakeEdge algorithm

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

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

3.4.1 LIMDD canonical form

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

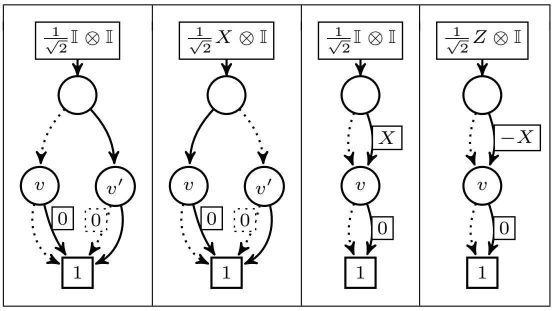
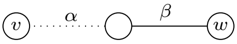
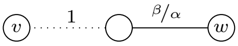

$\frac{1}{\sqrt{2}}\mathbb{I}\otimes\mathbb{I}$
v
v'
0
0
1
$\frac{1}{\sqrt{2}}X\otimes\mathbb{I}$
v
v'
0
0
1
$\frac{1}{\sqrt{2}}\mathbb{I}\otimes\mathbb{I}$
X
v
0
1
$\frac{1}{\sqrt{2}}Z\otimes\mathbb{I}$
$-X$
v
0
1

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

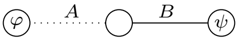
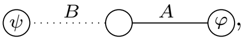
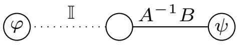
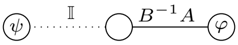
algorithms always construct the diagram in the backtrack of the recursion using a typical 'MakeNode' procedure for constructing canonical nodes [124], like in Sec. 3.3.3.) For instance, a QMDD node  with $\alpha, \beta \in \mathbb{C} \setminus \{0\}$ can be *reduced* into a canonical node by dividing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

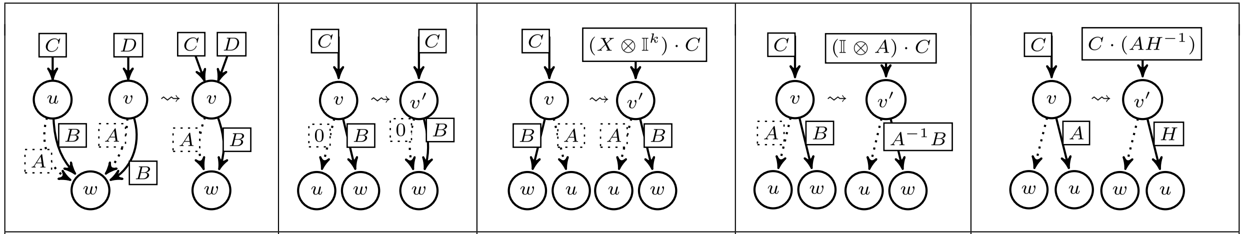
For LIMDD, we use a similar approach of dividing out 'common LIM factors.' However, we need to do additional work to obtain a unique high edge label (${}^{\beta}/_{\alpha}$ in the example above), as the PauliLIM group is more complicated than the group of complex numbers (scalars).

Definition 3.5 gives reduction rules for LIMDDs and Figure 3.11 illustrates them. The merge (1) and low factoring (4) rules fulfill the same purpose as in the QMDD case discussed above. In a Pauli-LIMDD, we may always swap high and low edges of a node v by multiplying the root edge LIM with $X \otimes \mathbb{I}$, as illustrated in Figure 3.10. The low precedence rule (3) makes this choice deterministic, but only in case $\mathsf{low}(v) \neq \mathsf{high}(v)$. Next, the zero edges (2) rule handles the case when α ór β are zero in the above, as in principle a edge e with label 0 could point to any node on the next level k, as this always yields a 0 vector of length 2^k (see semantics below Definition 3.2). The rule forces $\mathsf{low}(v) = \mathsf{high}(v)$ in case either edge has a zero label. We explain the interaction among the zero edges (2), low precedence (3) and low factoring (4) rules below. Finally, the high determinism rule (5) defines a deterministic function to choose LIMs on high edges, solving the most challenging problem of uniquely selecting a LIM on the high edge. We give an $\mathcal{O}(n^3)$ algorithm for this function in Sec. 3.4.2.

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

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

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



(1) Merge u, v into v

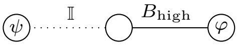
(2) Zero edges

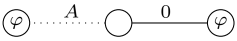
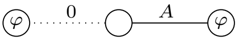
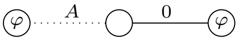
(3) Low precedence with $u \preccurlyeq w$

(4) Low factoring

(5) High determinism $H = \mathsf{HighLabel}(v)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

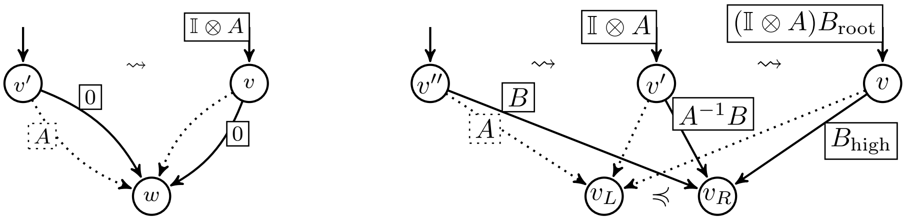


Figure 3.12: Reduced node construction in case $|\varphi_1\rangle = 0$ (left), and $|\varphi_0\rangle, |\varphi_1\rangle \neq 0$ and $v_L \preccurlyeq v_R$ (right). Not shown: for cases $|\varphi_0\rangle = 0$ and $v_R \preccurlyeq v_L$, we take instead root edge $X \otimes A$ and swap low/high edges.

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

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

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

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

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

v_M, we obtain,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.2 The MakeEdge subroutine: Maintaining canonicity during simulation

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

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

- First it ensures low precedence, switching e_0 and e_1 if necessary at Line 3. This is also done if e_0's label A is 0 to allow for low factoring (avoiding divide by zero).

- Low factoring, i.e., dividing out the LIM A, placing it on the root node, is visualized in Figure 3.12 for the cases $e_1 = 0/e_1 \neq 0$, and done in the algorithm at Line 6,7 / 9,11.

- The zero edges rule is enforced in the $B = 0$ branch by taking $v_1 := v_0$.

- The canonical high label B_{high} is computed by GetLabels, discussed below, for the semi-reduced node $\textcircled{$v_0$} \overset{\mathbb{I}}{\cdots\cdots} \textcircled{w} \xrightarrow{\hat{A}} \textcircled{v_1}$ with $v_0 \neq v_1$. With the resulting high label, it now satisfies the high determinism rule of Definition 3.5 with $\mathsf{HighLabel}(w) = B_{\text{high}}$.

- Finally, we merge nodes by creating an entry $(v_0, B_{\text{high}}, v_1)$ in a table called the *unique table* [59] at Line 13.

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

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

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

3.4.2.1 Choosing a canonical high-edge label

In order to choose the canonical high edge label of node v, the MakeEdge algorithm calls GetLabels (Line 10 of Algorithm 11). The function GetLabels returns a uniquely chosen LIM B_{high} among all possible high-edge labels which yield LIMDDs representing states that are Pauli-isomorphic to $|v\rangle$. We sketch the algorithm for GetLabels here and provide the algorithm in full detail in Section 3.5 (Algorithm 12). First, we characterize the eligible high-edge labels. That is, given a semi-reduced node $\textcircled{$v_0$}\overset{\mathbb{I}}{\cdots\cdots}\textcircled{v}\overset{\hat{A}}{\overline{\qquad}}\textcircled{v_1}$, we characterize all C such that the node $\textcircled{$v_0$}\overset{\mathbb{I}}{\cdots\cdots}\bigcirc\overset{C}{\overline{\qquad}}\textcircled{v_1}$ is isomorphic to $\textcircled{$v_0$}\overset{\mathbb{I}}{\cdots\cdots}\textcircled{v}\overset{\hat{A}}{\overline{\qquad}}\textcircled{v_1}$. Our characterization shows that, modulo some complex factor, the eligible labels C are of the form

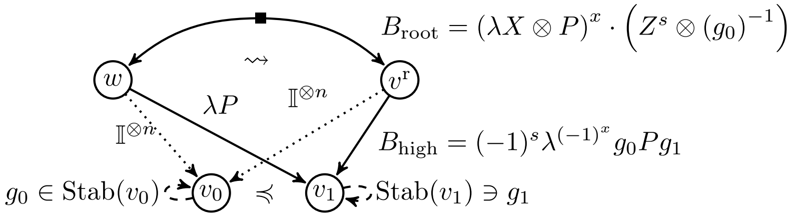
$$C \propto g_0 \cdot \hat{A} \cdot g_1, \quad \text{for } g_0 \in \mathrm{Stab}(|v_0\rangle), g_1 \in \mathrm{Stab}(|v_1\rangle) \tag{3.12}$$

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

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

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

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


$B_{\text{root}} = (\lambda X \otimes P)^x \cdot \left(Z^s \otimes (g_0)^{-1}\right)$
$\rightsquigarrow$
w
v^{r}
λP
$\mathbb{I}^{\otimes n}$
$\mathbb{I}^{\otimes n}$
$B_{\text{high}} = (-1)^s \lambda^{(-1)^x} g_0 P g_1$
$g_0 \in \text{Stab}(v_0)$
v_0
$\preccurlyeq$
v_1
$\text{Stab}(v_1) \ni g_1$

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

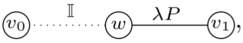
Figure 3.13: Illustration of finding a canonical high label for a semi-reduced node w, yielding a reduced node v^{r}. The chosen high label is the minimal element from the set of eligible high labels based on stabilizers g_0, g_1 of v_0, v_1 (drawn as self loops). The minimal element holds a factor $\lambda^{(-1)^x}$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

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

3.4.2.2 Checking whether two LIMDDs are Pauli-equivalent

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

3.5 Efficient algorithms for choosing a canonical high label

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

This section is structured as follows. In Sec. 3.5.1, we identify the canonical high label that we are after – the minimum element of a certain set – and presents an algorithm GetLabels, which finds it. The algorithm GetLabels calls several subroutines, which are presented in Sec. 3.5.2, Sec. 3.5.3 and Sec. 3.5.4. present algorithms which find this canonical label. Sec. 3.5.2 presents algorithms operating on group of Pauli operators.

3.5.1 Choosing a canonical high label

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

Figure 3.13 illustrates this process. In the figure, the left node w summarizes the status of the MakeEdge algorithm on Line 10, when this algorithm has enough information to construct the semi-reduced node $\textcircled{v_0}\overset{\mathbb{I}^{\otimes n}}{\cdots\cdots}\textcircled{w}\overset{\lambda P}{\rule{2em}{0.4pt}}\textcircled{v_1}$, shown on the left. The node v^r, on the right, is the canonical node, and is obtained by replacing w's high edge's label by the canonical label B_{high}. This label is chosen by minimizing the expression $B_{\text{high}} = (-1)^s \lambda^{(-1)^x} g_0 P g_1$, where the minimization is over $s, x \in \{0,1\}, g_0 \in \mathit{Stab}(|v_0\rangle), g_1 \in \mathit{Stab}(|v_1\rangle)$, subject to the constraint that $x = 0$ if $v_0 \neq v_1$. We have $|w\rangle \simeq_{\text{Pauli}} |v^r\rangle$ by construction as intended, namely, they are related via $|w\rangle = B_{\text{root}} |v^r\rangle$. Theorem 3.8 shows that this way to choose the high label indeed captures all eligible high labels, i.e., a node $\textcircled{v_0}\overset{\mathbb{I}}{\cdots\cdots}\textcircled{v^r}\overset{B_{\text{high}}}{\rule{2em}{0.4pt}}\textcircled{v_1}$ is isomorphic to $|w\rangle$ if and only if B_{high} is of this form.

Theorem 3.8 (Eligible high-edge labels)**.** Let $\textcircled{v_0}\overset{\mathbb{I}^{\otimes n}}{\cdots\cdots}\textcircled{w}\overset{\lambda P}{\rule{2em}{0.4pt}}\textcircled{v_1}$ be a semi-reduced n-qubit node in a Pauli-LIMDD, where v_0, v_1 are reduced, P is a Pauli string and $\lambda \neq 0$. For all nodes $v = \textcircled{v_0}\overset{\mathbb{I}^{\otimes n}}{\cdots\cdots}\textcircled{v}\overset{B_{\text{high}}}{\rule{2em}{0.4pt}}\textcircled{v_1}$, it holds that $|w\rangle \simeq |v\rangle$ if and only if

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

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

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

Proof. It is straightforward to verify that the isomorphism B_{root} in eq. (3.14) indeed maps $|w\rangle$ to $|v\rangle$ (as $x = 1$ implies $v_0 = v_1$), which shows that $|w\rangle \simeq |v\rangle$. For the converse direction, suppose there exists an n-qubit Pauli LIM C such that $C|w\rangle = |v\rangle$, i.e.,

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

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

3.5.2 Efficient linear-algebra algorithms for stabilizer subgroups

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

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

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

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

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

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

$$\begin{pmatrix} X & \otimes & X & \otimes & X \\ \mathbb{I} & \otimes & Z & \otimes & Y \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & | & 0 & 0 & 0 \\ 0 & 0 & 1 & | & 0 & 1 & 1 \end{pmatrix}.$$

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

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

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

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

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

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

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

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

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

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

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

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

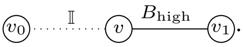
$$\begin{pmatrix} 3X & \otimes & X & \otimes & X \\ -\frac{1}{2}i\mathbb{I} & \otimes & Z & \otimes & Y \end{pmatrix} \sim \begin{pmatrix} 1 & 1 & 1 & | & 0 & 0 & 0 & | & 3 & 0 \\ 0 & 0 & 1 & | & 0 & 1 & 1 & | & \frac{1}{2} & \frac{3\pi}{2} \end{pmatrix}$$

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

Finally, we emphasize that the algorithms above rely on bitwise xor-ing, which is a commutative operation. Since conventional (i.e., factor-respecting) multiplication of Pauli operators is not commutative, the algorithms above are not straightforwardly applicable to arbitrary PAULILIM$_n$ input. (When the input consist of pairwise commuting Pauli operators, such as stabilizer subgroups [3], the algorithms can be made to work by adjusting row addition to keep track of the scalar.) Fortunately, since Pauli strings either commute or anti-commute, row addition may only yield factors up to the ± sign, not the resulting Pauli strings. This feature, combined with the stipulated order assigning least significance to the factor, enables us to invoke the algorithms above as subroutine. We do so in Sec. 3.4.2.1 and Sec. 3.5.3.

3.5.3 Constructing the stabilizer subgroup of a LIMDD node

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

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

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

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

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

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

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

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

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 π. Lemma 3.1 shows that such an isomorphism set can be expressed in terms of the stabilizer group of $|v\rangle$.

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

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 σ 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 $\sigma = \pi(\pi^{-1}\sigma) \in \pi \cdot \text{Stab}(|\varphi\rangle)$. □

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

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

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

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

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

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

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

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

---

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

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

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

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

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

Lemma 3.2. The coset $S := \langle G_0\rangle \cap \pi_1^{-1}\pi_0 \cdot \langle G_1\rangle$ is nonempty if and only if the lexicographically smallest element of the set $S = \pi_0^{-1}\pi_1 \langle G_1\rangle \cdot \langle G_0\rangle = \left\{\pi_0^{-1}\pi_1 g_1 g_0 \mid g_0 \in G_0, g_1 \in G_1\right\}$ is $1 \cdot \mathbb{I}$.

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

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

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

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

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

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

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

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

3.5.4 Efficiently finding a minimal LIM by multiplying with stabilizers

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

Lemma 3.4. Let v_0 and v_1 be LIMDD nodes, R a Pauli string and $\nu, \nu' \in \mathbb{C}$. Define $G = \text{Stab}(v_0) \cup \text{Stab}(v_1)$. 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 λg as $g = (-1)^x g_0 g_1$ and $\lambda g = (-1)^y h_0 h_1$ for some $x, y \in \{0, 1\}$ and $g_0, h_0 \in \text{Stab}(v_0)$ and $g_1, h_1 \in \text{Stab}(v_1)$. Combining these

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

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

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

```
output: Pauli LIM $\pi$, stabilizer subgroup generating set $G$ s.t. $\pi\langle G\rangle = \pi_0\langle G_0\rangle \cap \pi_1\langle G_1\rangle$
procedure IntersectIsomorphismSets(stabilizer subgroup generating sets $G_0, G_1$, Pauli-LIMs $\pi_0, \pi_1$)
    $\pi := LexMin(G_0, G_1, \pi_1^{-1}\pi_0)$
    if $\pi = \mathbb{I}$ then
        $(g_0, g_1) = ArgLexMin(G_0, G_1, \pi_1^{-1}\pi_0)$
        $\pi := \pi_0 \cdot g_0$
        $G := IntersectStabilizerGroups(G_0, G_1)$
        return $(\pi, G)$
    else
        return None
```

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

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

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

```
procedure GetIsomorphism(Edge $\xrightarrow{A}(v)$, $\xrightarrow{B}(w)$ with $v, w$ reduced, $A \neq 0 \lor B \neq 0$)
    if $v = w$ and $A, B \neq 0$ then
        return $B \cdot A^{-1}$
    return None
```

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

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

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

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

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

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

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

3.6 Numerical search for the stabilizer rank of Dicke states

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

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

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

Since our heuristic algorithm finds only an upper bound on the stabilizer rank, and not a lower bound, by construction we cannot guarantee any statement on the scaling of the rank itself. However, our approach could have found only stabilizer 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. [64], which has been explained in more detail in [76]. The algorithm follows a simulated annealing approach: on input n, w and χ, it performs a random walk through sets of χ stabilizer states. It starts with a random set V of χ stabilizer states on n qubits. In a single 'step', the algorithm picks one of these states $|\psi\rangle \in V$ at random, together with a random n-qubit Pauli operator P, and replaces the state $|\psi\rangle$ with $|\psi'\rangle := c(\mathbb{I} + P)\,|\psi\rangle$ with c a normalization constant (or repeats if $|\psi'\rangle = 0$), yielding a new set V'. The step is accepted with certainty if $F_V < F_{V'}$, where $F_V := |\langle D_w^n|\,\Pi_V\,|D_w^n\rangle|$ with Π_V the projector on the subspace of the n-qubit Hilbert space spanned by the stabilizer states in V. Otherwise, it is accepted with probability $\exp(-\beta(F_{V'} - F_V))$, where β should be interpreted as the inverse temperature. The algorithm terminates if it finds $F_V = 1$, implying that $|D_w^n\rangle$ can be written as linear combination of V, outputting the number χ as (an upper bound on) the stabilizer rank of $|\psi\rangle$. For a fixed χ, we use identical values to Bravyi et al. [64] and vary β from 1 to 4000 in 100 steps, performing 1000 steps at each value of β.

3.7 Related work

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

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

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

Table 3.2: Heuristically-found upper bounds on the stabilizer rank χ of Dicke states $|D^n_w\rangle$ (eq. (3.21)) using the heuristic algorithm from Bravyi et al. [64], see text in Section 3.6 for details. We investigated up to 9 qubits using the SURF supercomputing cluster (approximately the same as the number of qubits reached in the literature as described in the text). Empty cells indicate non-existing or not-investigated states. In particular, we have not investigated $w > \lfloor \frac{n}{2} \rfloor$ since $|D^n_w\rangle$ 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.

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

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

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

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

3.8 Discussion

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

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

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

Since we know from experience that implementing a decision diagram framework is a major endeavor, we leave an implementation of the Pauli-LIMDD, in order to observe its runtimes in practice on relevant quantum circuits, to future work. We emphasize that from the perspective of algorithm design, we have laid all the groundwork for such an implementation, including the key ingredient for the efficiency of many operations for existing decision diagrams: the existence of a unique canonical representative of the represented function, combined with a tractable MakeEdge algorithm to find it.

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

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

Chapter 4

Efficient implementation of LIMDDs for Quantum Circuit Simulation

Realizing the promised advantage of quantum computers over classical computers requires both physical devices and corresponding methods for the design, verification and analysis of quantum circuits. In this regard, decision diagrams have proven themselves to be an indispensable tool due to their capability to represent both quantum states and unitaries (circuits) compactly. Nonetheless, recent results show that decision diagrams can grow to exponential size even for the ubiquitous stabilizer states, which are generated by Clifford circuits. Since Clifford circuits can be efficiently simulated classically, this is surprising. Moreover, since Clifford circuits play a crucial role in many quantum computing applications, from networking, to error correction, this limitation forms a major obstacle for using decision diagrams for the design, verification and analysis of quantum circuits. The *Local Invertible Map Decision Diagram* (LIMDD), introduced in Chapter 3, solves this problem by combining the strengths of decision diagrams and the stabilizer formalism that enables efficient simulation of Clifford circuits. However, LIMDDs have only been introduced on paper thus far and have not been implemented yet—preventing an investigation of their practical capabilities through experiments. In this work, we present the first implementation of LIMDDs for quantum circuit simulation. The resulting package is available under a free license at `https://github.com/cda-tum/ddsim/tree/limdd`.

By implementing the LIMDD as a software package, this chapter contributes to answering Research Question 1:

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

This chapter answers this question by empirically testing to what extent LIMDDs simulate quantum circuits that involve stabilizer states more effectively than existing DDs. In other words, this chapter assesses to what extent LIMDDs constitute an answer to Research Question 1. A case study confirms the improved performance when simulating the Quantum Fourier Transform applied to a stabilizer state.

4.1 Introduction

Quantum computing is a new and drastically different computing paradigm promising to solve certain problems that are intractable for classical computers. Examples of such problems include unstructured search [12, 140, 231], integer factorization [291] and quantum chemistry [196]. This computational power is harnessed by using quantum mechanical effects such as *superposition*, where the system can be in a linear combination of multiple states, and *entanglement*, where operations on one part of the system can affect other parts as well. In the near term, quantum computers classified as *Noisy Intermediate-Scale Quantum* (NISQ) devices are expected to both deliver empirical evidence of quantum advantage over classical computers as well as solve practical problems. However, the ability to build large quantum computers is not by itself sufficient if there are no means of harnessing their power: we also need tools for the design of quantum circuits, i.e., for simulation, compilation, and verification. The *classical simulation* of quantum computers in particular has been used in service of the verification of quantum algorithms [71, 73, 75], and is a way to quantify "the elusive boundary at which a quantum advantage may materialize" [77].

A major challenge in the classical design of quantum systems is that the memory requirements grow exponentially in the number of qubits. Contrary to the classical world, where representing a system state with m classical bits requires only a linear amount of memory, the state of an n-qubit quantum system is described by a vector of 2^n complex numbers. Current estimates indicate that at least hundreds of qubits are required to perform useful tasks on a quantum computer [145]. However, even current super-computing clusters can only handle systems with between 50 and 60 qubits represented as vectors [173]. Therefore, dedicated data structures and design methods which can tackle the exponential complexity of quantum computing need to be developed.

Given that merely representing a quantum state may require an exponential amount of memory with respect to the number of qubits, it comes as no surprise that conducting quantum circuit simulation, for example, is a hard problem. Even more dauntingly, verification of quantum circuits at its heart considers quantum operations and therefore has $2^n \times 2^n$ complexity when implemented naively. Fortunately, due to the characteristics of quantum computing, quantum circuit simulation can help to verify the equivalence of two circuits to a very high degree of confidence, despite the infinite number of possible input states. More precisely, providing an appropriate quantum state as input to the circuits under consideration and checking equivalence of the resulting states will provide a counterexample for non-equivalent circuits with a high probability [75]. Selecting basis states as input, i.e., states without superposition or entanglement, does not always suffice for this purpose, but random *stabilizer states*, introduced next, have been shown to do the trick [71]. This makes quantum circuit simulation a key component of design automation for quantum computing and hence the subject of the current chapter.

Stabilizer states are ubiquitous in quantum computing. They are computed by so-called Clifford circuits, a subset of the universal quantum computing gate set [3, 135]. For example, stabilizer states include the Bell state and GHZ state. Further, Clifford circuits play an essential role in error correction [135, 137], entanglement distillation [39] and are used in one-way quantum computing [66]. Any n-qubit stabilizer state can be represented using memory in the order of $\mathcal{O}(n^2)$ and the non-universal fragment of Clifford circuits can be simulated efficiently by manipulating this representation [3, 135]. In fact, stabilizer states capture the essential symmetries of all universal quantum computing states, which is why they also play a key role in the reduction from verification to simulation explained above. For these reasons, it can be argued that any practically efficient classical simulation of (universal) quantum computing should also support Clifford circuits and the stabilizer states they generate. The current work removes this limitation from existing (universal) simulation approaches based on decision diagrams.

Decision diagrams (DDs) are a tried-and-tested data structure in the world of classical design automation [67, 69, 103, 229, 328]. They have also shown promising results in quantum design automation [4, 242, 332, 337, 342, 369]. Decision diagrams exploit redundancies in the state vector and operations matrix to enable a compact representation in many cases. Unfortunately, a state-of-the art quantum simulation method called Quantum Multi-valued Decision Diagram (QMDD) [227] does not efficiently represent stabilizer states [337], which poses a serious bottleneck to their adoption, as explained above, but also observed in practice [71]. We introduced the *Local Invertible Map Decision Diagram* (LIMDD, [337]) in Chapter 3, which addresses this shortcoming. LIMDDs efficiently represent stabilizer states, they simulate Clifford circuits in polynomial time and can efficiently apply many Clifford gates also to non-stabilizer states. However, LIMDDs lack an implementation to demonstrate that their asymptotic advantage also translates to practical use cases.

In this chapter, we present an implementation of LIMDDs for universal simulation of quantum circuits (and thus design automation) based on the QMDD package [242,369]. We adapt techniques that are tried and tested in the implementations of both classical and quantum decision diagram packages, and enrich them with special considerations to efficiently handle *Local Invertible Maps* (LIMs). For the first time, this leads to an implementation that realizes LIMDDs, and also demonstrates the potential of LIMDDs. In particular, we show their use for verification through circuit equivalence checking for a case study on the Quantum Fourier Transform (QFT, [176, 240]). The results confirm that the more complex LIMDD-based simulator surpasses a state-of-the-art decision-diagram-based simulator for larger instances. The resulting implementation is available at `https://github.com/cda-tum/ddsim/tree/limdd` under the MIT license.

The remainder of this chapter is structured as follows. Section 4.2 briefly reviews the necessary background on quantum computing and classical quantum circuit simulation. In Section 4.3, we briefly review existing decision diagrams for quantum computing and motivate the need for an efficient LIMDD implementation. Section 4.4 details the techniques used to enable efficient construction and manipulation of LIMDDs. In Section 4.5, we provide an experimental evaluation showcasing the performance of the proposed implementation. Finally, Section 4.7 concludes the chapter.

4.2 Background

To keep this chapter self-contained, this section provides the necessary background on quantum computing as well as classical quantum circuit simulation. For a more elaborate introduction to quantum computing, we refer the reader to Section 2.2; for an introduction to decision diagrams, see Section 2.3.

4.2.1 Quantum states and operations

The basic unit of information in quantum computing is the *quantum bit* or *qubit* [240]. A single-qubit quantum state $|\psi\rangle$ can be described by its amplitude vector $|\psi\rangle = \alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$ with complex amplitudes $\alpha_0, \alpha_1 \in \mathbb{C}$. Here, $|0\rangle = [\begin{smallmatrix}1\\0\end{smallmatrix}], |1\rangle = [\begin{smallmatrix}0\\1\end{smallmatrix}]$ denote the two basis states and are analogous to the basis states 0 and 1 in classical computing, in the sense that a classical register containing one bit can be either in state 0 or 1. An amplitude vector must meet the normalization constraint $|\alpha_0|^2 + |\alpha_1|^2 = 1$. If both amplitudes α_0 and α_1 are non-zero, the state is in *superposition*. For a state $|\varphi\rangle$, we write $\langle\varphi| = |\varphi\rangle^\dagger$ to denote its conjugate transpose. Two quantum states $|\varphi\rangle, |\psi\rangle$ are considered equal if the absolute value of their in-product equals one, i.e., $|\langle\varphi| \cdot |\psi\rangle| = 1$, also written as $|\langle\varphi|\psi\rangle| = 1$. We say that $|\varphi\rangle, |\psi\rangle$ are approximately the same state if $|\langle\varphi|\psi\rangle| \approx 1$. When measuring a state, the probability that a given basis state is the outcome is the squared magnitude of the amplitude of that basis state, i.e, for the state $\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$, the probability of measuring the zero state is $|\alpha_0|^2$. Therefore, in a physical quantum computer, the individual amplitudes are fundamentally non-observable and information about the quantum state can only be extracted through destructive measurement, i.e., after measurement, superposition is destroyed.

A quantum register may consist of multiple qubits. A register $|\varphi\rangle$ consisting of n qubits has 2^n basis states $|i\rangle$ with $i \in \{0,1\}^n$, each with a corresponding amplitude α_i, i.e., $|\varphi\rangle = \sum_i \alpha_i |i\rangle$. Here a basis state $|i\rangle$ with i a bit string $b_1 b_2 ... b_n$ is formed from the single qubit basis vectors above using tensor products $|b_1\rangle \otimes |b_2\rangle \otimes ... \otimes |b_n\rangle$, written shortly as $|b_1\rangle |b_2\rangle ... |b_n\rangle = |b_1 b_2 ... b_n\rangle = |i\rangle$. Here the *tensor product* of a $k \times \ell$ matrix $A = (a_{ij})_{ij}$ and an $n \times m$ matrices $B = (b_{ij})_{ij}$ is the $kn \times \ell m$ matrix $C = (a_{ij} b_{xj})_{ijxy}$. Alternatively, $|\varphi\rangle$ can be understood as the pseudo-Boolean function $f \colon \{0,1\}^n \to \mathbb{C}$, defined as $f(i) = \langle i|\varphi\rangle = \alpha_i$. The normalization constraint is generalized to $\sum_{0 \leq i < 2^n} |\alpha_i|^2 = 1$. A quantum state $|\varphi\rangle$ is *entangled* if it cannot be written as a tensor product of single qubit states, i.e, $|\varphi\rangle = |\varphi_1\rangle \otimes \cdots \otimes |\varphi_n\rangle$.

Example 4.1. Consider the quantum state $1/\sqrt{2} \cdot (|00\rangle + |11\rangle)$, commonly known as the *Bell state* [240]. As a vector, it would be written as $1/\sqrt{2} \cdot [\,1\ 0\ 0\ 1\,]^{\mathrm{T}}$. If this state is measured, we have equal probabilities of obtaining as outcome one of the basis states $|00\rangle$ and $|11\rangle$, and zero probability of seeing the states $|01\rangle$ and $|10\rangle$.

In addition to superposition, this quantum state shows *entanglement*. Measuring a value for one qubit of the Bell state would immediately fix the value of the other qubit corresponding to the measurement outcome, e.g., after measuring $q_1 = |0\rangle$ (or $q_1 = |1\rangle$) we immediately know that $q_0 = |0\rangle$ (or $q_0 = |1\rangle$).

Quantum states are manipulated through quantum gates. A quantum gate is any linear operator mapping quantum states to quantum states, i.e., a unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$, where n is the number of qubits. A quantum algorithm consists of a series of gates applied sequentially, e.g., $U = U_m \cdots U_1$ is an algorithm consisting of m gates, which first applies U_1, then U_2, up to U_m. Thus, U denotes the unitary matrix corresponding to applying all the gates. If a quantum state $|\varphi\rangle$ serves as input to U, then the output is the quantum state $U \cdot |\varphi\rangle$. We say that a quantum algorithm $U_1, \ldots, U_m$ is equivalent to another quantum algorithm $V_1, \ldots, V_\ell$ iff they effect the same unitary matrix, i.e., if $U_m \cdots U_1 = V_\ell \cdots V_1$.

Example 4.2. Three examples of common quantum gates are the single-qubit phase-shift operation S, the single-qubit Hadamard operation H, and the two-qubit controlled-NOT operation $CNOT$ (here shown with control on the first qubit, target on the second qubit).

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The state from Example 4.1 can be created by starting the $|00\rangle$ basis state and then applying a Hadamard operation on the first qubit, followed by a controlled-NOT. Using the tensor product for parallel composition of gates as usual [240], this can be written as $CNOT \times (H \otimes \mathbb{I}) \times |00\rangle = |11\rangle$. Figure 4.1 shows another example of a 3-qubit quantum circuit built from these three gates generalized to multiple qubits. The circuit is to be read from left to right, so that the Hadamard gate is the first gate, applied to qubit q_2. The $\bullet$ denotes the control qubit of a $CNOT$ gate; the $\oplus$ denotes

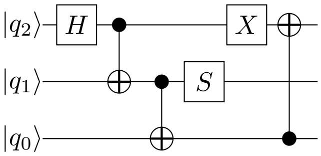


Figure 4.1: A quantum circuit on three qubits, q_2–q_0.

its target qubit.

An important, though non-universal, subset of quantum circuits are Clifford circuits [137], which consist only of the Clifford gates S, H and $CNOT$. Clifford circuits are ubiquitous in quantum computing because they represent the symmetries that occur in all quantum states albeit they cannot generate arbitrary transformations (hence they are non-universal). As a consequence, they play an essential role in error correction [135, 137], entanglement distillation [39] and are used in one-way quantum computing [66]. Clifford circuits are intimately related to the Pauli gate set:

$$\mathbb{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Example 4.3. The circuit shown in Figure 4.1 is a Clifford circuit, since it only consists of Clifford operations and gates that can be built from Clifford operations. The Pauli gates can be built from Clifford gates, namely as $Z = S^2$ and $X = H \times Z \times H$, and $Y = iXZ$; thus, Pauli gates are Clifford gates.

4.2.2 Classical quantum circuit simulation

The classical simulation of a quantum circuit is the process of simulating a quantum circuit on a classical binary computer. It is an important task in the context of quantifying the capability of physical quantum computers and in the development of quantum algorithms.

Circuit simulation can be conducted in a straightforward fashion by repeated matrix-vector multiplication. The simulation starts with an initial state and applies the quantum gates one after the other. Each quantum operation is represented by a unitary matrix U_t of dimension $2^n \times 2^n$ and each quantum state by a unit vector $|\varphi_t\rangle$ of dimension 2^n. The evolution of a state at time step t is then given by $|\varphi_{t+1}\rangle = U_{t+1} |\varphi_t\rangle$

(with $|\varphi_0\rangle$ commonly being the vector representing the all-zero state, $|\varphi_0\rangle = |0\ldots0\rangle$).

Clifford circuits can be efficiently simulated on a classical computer [3], which is surprising given their importance, but not a contradiction given their non-universality. Starting from the all-zero state $|0^n\rangle$, Clifford circuits only yield so-called *stabilizer states*. An n-qubit stabilizer state $|\varphi\rangle$ can be uniquely specified by the set of Pauli operators $G = \pm P_1 \otimes \ldots \otimes P_n$ with $P_i \in \{\mathbb{I}, X, Y, Z\}$ that stabilize it, i.e., which satisfy $G\,|\varphi\rangle = |\varphi\rangle$. This set forms an abelian group, and can always be described succinctly by a set of n generators $G_1, \ldots, G_n \in \pm\{\mathbb{I}, X, Y, Z\}^n$. We can thus think of this generator set as an $n \times n$ matrix of local Pauli operators, where each row (a generator) also has an additional plus or minus sign as shown in Example 4.4. This characterization is the key to efficiently classical simulation of stabilizer states [135] since the Clifford gates can be applied directly to the generator set describing the state.

Example 4.4. The three-qubit Clifford circuit in Figure 4.1 can be simulated for time steps $t = 0, 1, 2$ using explicit vector representation as follows.

$$\begin{aligned}
|\varphi_0\rangle &= |000\rangle && \text{apply } H \text{ on } q_2 \rightarrow\\
|\varphi_1\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|001\rangle && \text{apply CNOT on control } q_1 \text{ and target } q_2 \rightarrow\\
|\varphi_2\rangle &= \frac{1}{\sqrt{2}}|000\rangle + \frac{1}{\sqrt{2}}|011\rangle && \text{(which is } |0\rangle \otimes \text{Bell state)}
\end{aligned}$$

Alternatively, we may represent each $|\varphi_t\rangle$ as a generator set $G(\varphi_t) = \{G_1, G_2, G_3\}$.

$$\begin{aligned}
G(\varphi_0) &= \{Z\,\mathbb{I}\,\mathbb{I},\ \mathbb{I}\,Z\,\mathbb{I},\ \mathbb{I}\,\mathbb{I}\,Z\} && \text{apply } H \text{ on } q_2 \rightarrow\\
G(\varphi_1) &= \{Z\,\mathbb{I}\,\mathbb{I},\ \mathbb{I}\,Z\,\mathbb{I},\ \mathbb{I}\,\mathbb{I}\,X\} && \text{apply CNOT on control } q_1 \text{ and target } q_2 \rightarrow\\
G(\varphi_2) &= \{Z\,\mathbb{I}\,\mathbb{I},\ \mathbb{I}\,ZZ,\ \mathbb{I}\,XX\}
\end{aligned}$$

We call the generator set representing a stabilizer state a stabilizer tableau. The *stabilizer formalism* stipulates how the tableau should be modified for different Clifford gates [3, 135] as exemplified in Example 4.4. It forms a non-universal, but classically tractable region of quantum computing, whereas decision diagrams considered in this work target universal quantum computing. Nonetheless, Clifford circuits and stabilizer states are important in many domains, as discussed in the introduction.

4.2.3 Verification of quantum circuits

A popular similarity metric for comparing two quantum circuits U, V is the *average fidelity*, $\mathcal{F}_{\text{avg}}(U, V) = \frac{1}{1+2^n}(1 + |\text{tr}(UV^\dagger)|^2)$, where $\text{tr}(M)$ denotes the trace of M. This metric has value 1 iff $U = V$, and has $\mathcal{F}_{\text{avg}}(U, V) < 1$ otherwise. Burgholzer et. al [71], building on a result from Kueng and Gross [190] showed how the average fidelity relates to inputs with random stabilizer states:

Theorem 4.1 (Burgholzer et. al [71])**.** Suppose $|g\rangle$ is a random stabilizer state, and U, V are unitary matrices. Then there is the following relationship between the expectation value and the average fidelity:

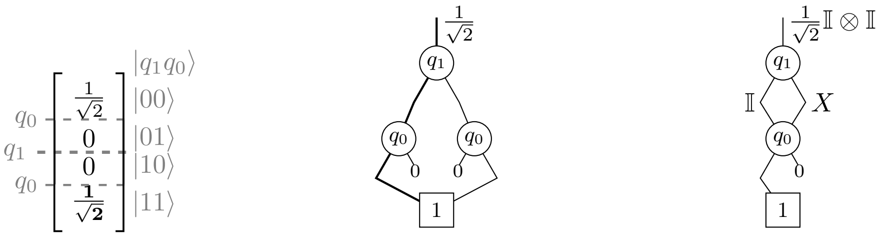
$$\mathbb{E}_{|g\rangle}[\langle g| V^\dagger \cdot U |g\rangle] \approx \mathcal{F}_{\text{avg}}(U, V) \tag{4.1}$$

Consequently, the average fidelity $\mathcal{F}_{\text{avg}}(U, V)$ can be approximated by simulating the two circuits on several random stabilizer states. Put another way, this quantifies the statement that a random stabilizer state is to a quantum circuit what a random input is to a classical circuit. This motivates the approach of Burgholzer et al.: they repeatedly generate (two copies of) a random stabilizer state $|g\rangle$ as input, then they classically simulate the two circuits on this input, obtaining states $U|g\rangle, V|g\rangle$. Lastly, they compute the inner product of the output states $\langle g| V^\dagger \cdot U |g\rangle$; if the absolute magnitude of this number is smaller than 1, then $|g\rangle$ is a counterexample which certifies that $U \neq V$.

We use this reduction from circuit verification to circuit simulation as the motivation for the setup in our case study in Section 4.5. We provide a brief overview of methods for the verification of quantum algorithms and circuits in Section 2.4.

4.3 Motivation

Efficient classical simulation of quantum circuits is an important task for the development of both quantum circuits and their compilation toolchains [75, 77]. As reviewed above, this task is conceptually simple (matrix-vector multiplication), but practically hard due to the memory requirements of classical descriptions of quantum states, i.e., state vectors require an exponential amount of memory with respect to the number of qubits. However, certain families of quantum circuits, such as those consisting



(a) Vector (b) QMDD (c) Pauli-LIMDD

Figure 4.2: Different representations of a Bell state

only of Clifford gates, can be simulated in polynomial time by the stabilizer formalism that exploits the strong algebraic structure present in stabilizer states [3, 135]. These techniques have the disadvantage that they do not encompass all of quantum computing, since they cannot produce all quantum gates, i.e., they are not *universal*. For general quantum circuits, i.e., those with no restrictions on the gate set, decision diagrams as reviewed later in this section are a promising data structure to drastically reduce memory requirements in many cases. However, the stabilizer formalism and decision diagrams have thus far excelled only in their respective areas. LIMDDs unite the capabilities of both worlds and thereby enable efficient representation of multiple classes of quantum states.

Although the verification of a given quantum circuit is likely more difficult than simulation of that circuit on a given input, Burgholzer et al. [71] recently showed that simulation can nevertheless be very useful for verification as explained in Sec. 4.2.3. In particular, they showed that a promising approach is to simulate the circuit on a certain state called a stabilizer state, and gave qualitative and quantitative analytical guarantees on the errors found by this method. Since LIMDDs excel on this family of quantum states, we adopted this approach for our case study in Section 4.5.

The remainder of this section first reviews the basics of QMDDs and LIMDDs, and then discusses their respective strengths (as far as they have been analytically investigated thus far). Based on this, we motivate the need for an implementation of LIMDDs.

4.3.1 Quantum Multiple-valued Decision Diagrams (QMDDs)

Representing quantum states and operations in a straightforward fashion as vectors and matrices requires an exponential amount of memory with respect to the number of qubits. Decision diagrams are an established data structure that approach this problem by providing a compact representation by exploiting redundancies in the data in many cases. There are multiple types of decision diagrams for the quantum domain [4, 242, 332, 337, 342]. We focus on *Quantum Multiple-valued Decision Diagrams* (QMDDs, [242, 374]) since they are the state of the art for decision diagram-based quantum circuit simulation.

Conceptually, the QMDD corresponding to the amplitude vector $|\varphi\rangle \in \mathbb{C}^{2^n}$ can be built as follows. First, we repeatedly split the amplitude vector in two equal halves, until the individual amplitudes are reached, thus obtaining a binary tree (of height n), in which a node at height k represents a (sub-)vector of length 2^k. Next, whenever two nodes represent states $|\varphi\rangle, |\psi\rangle$ satisfying $|\varphi\rangle = \lambda \cdot |\psi\rangle$ for some $\lambda \in \mathbb{C}$, we merge these two nodes (discarding one of the subtrees) and place the factor λ on one of the two incoming edges. This *reduced* QMDD is now a directed acyclic graph (DAG) and no longer a tree. Thus, the way the QMDD exploits structure in the vector is by recognizing repeated sub-vectors (or more precisely: sub-vectors which are equal up to a complex constant), which is how it can avoid the exponential blowup in many cases. Note that this construction is explained for illustration purposes only as working with decision diagrams does not at any point require explicitly storing the vector representing a quantum state. For a formal definition we refer to [227].

One can efficiently apply a given operation U to a state $|\varphi\rangle$ when both are given as QMDDs. This is done by the MULTIPLY algorithm, which recursively traverses the decision diagrams of U and $|\varphi\rangle$ and builds the DD corresponding to the state $|\psi\rangle = U \cdot |\varphi\rangle$. We briefly sketch how this algorithm works. First, note that if $|\varphi\rangle = |0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes |\varphi_1\rangle$, then the DD node representing $|\varphi\rangle$ has two children, v_0 and v_1, which represent the amplitude vectors $|v_j\rangle = |\varphi_j\rangle$ for $j = 0, 1$. Similarly, the matrix $U = \sum_{ij} |i\rangle \langle j| \otimes U_{ij}$ is represented by a QMDD node with four children v_{ij}, with each v_{ij} representing the submatrix U_{ij}, a quadrant of U. The algorithm first constructs decision diagrams for the states $U_{ij} \cdot |\varphi_j\rangle$ using four recursive calls to MULTIPLY(U_{ij}, v_j). Next, it construct decision diagrams representing the states $|\psi_i\rangle = U_{i0} |\varphi_0\rangle + U_{i1} |\varphi_1\rangle$ for $i = 0, 1$ using two calls to a procedure ADD implementing addition on QMDDs. Last, it makes a node whose two children are $|\psi_0\rangle$ and $|\psi_1\rangle$, obtaining a node representing $|\psi\rangle = |0\rangle \otimes |\psi_0\rangle + |1\rangle \otimes |\psi_1\rangle = U \cdot |\varphi\rangle$, as intended. We use dynamic programming to store the results of all intermediate, recursive calls to MULTIPLY; as is typically done to avoid the exponential-time behavior occurring when all paths in the DAG are considered. In light of this, we remark that in this work we consider matrices U representing a universal gate set that nonetheless each have a small number of nodes that scales as $\mathcal{O}(n)$.

Thus, while it is instructive to consider how a QMDD may be constructed from a given amplitude vector (in the way described above), our algorithms use a more efficient approach, never “expanding” the decision diagram to its amplitude vector, instead working directly on the DD representation of the vectors and matrices. Indeed, this is the primary strength of decision diagrams in general: that they can work on compressed data without decompressing it first.

Example 4.5. Consider the quantum state $1/\sqrt{2} \cdot (|00\rangle + |11\rangle)$ from Example 4.1. Figure 4.2a shows the corresponding vector, with superimposed information on the splitting by qubit (on the left) and the basis states to each amplitude (on the right). Figure 4.2b shows the same quantum state represented as QMDD.

Retrieving the amplitude of a given quantum state requires traversing the decision diagram and multiplying the edge weights along the way. For readability, edge weights of 1 are omitted; and edges with weight 0 are cut off and represented as stubs. The bolded path in Figure 4.2b represents the state $|00\rangle$, and following it gives $1/\sqrt{2} \cdot 1 \cdot 1 = 1/\sqrt{2}$.

While QMDDs enable compact representation of quantum states in many cases, Vinkhuijzen et al. [337] showed they can become exponentially sized for stabilizer states, which can be efficiently simulated classically using the stabilizer formalism as discussed in Sec. 2.2.1. These states are the intermediate states of circuits consisting of only Clifford gates, thus preventing QMDDs from simulating such circuits efficiently.

4.3.2 Local Invertible Map Decision Diagrams (LIMDDs)

LIMDDs remove the limitation of QMDDs that cannot efficiently represent every stabilizer state. They can represent each stabilizer state in polynomial space by not just merging nodes that are equivalent up to a scalar, but also those equivalent up to a LIM transformation while retaining universality. A LIM is similar to the stabilizer generator except that it includes an arbitrary scalar.

Definition 4.1 (Local Invertible Map (LIM), adapted from [337])**.** An n-qubit *Local Invertible Map* (LIM) is an operator P of the form $P = \lambda P_n \otimes \cdots \otimes P_1$, where the matrices $P_i \in \{\mathbb{I}, X, Y, Z\}$ are local Pauli matrices and $\lambda \in \mathbb{C} \setminus \{0\}$. An isomorphism between two n-qubit quantum states $|\varphi\rangle, |\psi\rangle$ is a LIM P such that $P|\varphi\rangle = |\psi\rangle$. We then say that $|\varphi\rangle$ is isomorphic to $|\psi\rangle$, denoted $|\varphi\rangle \simeq |\psi\rangle$. Note that isomorphism is an equivalence relation.

In fact, LIMDDs can efficiently apply most Clifford gates to any quantum state (i.e., even to non-stabilizer states), without increasing the size of the diagram by more than a factor two. LIMDDs extend QMDDs by annotating an edge not only with a complex-valued weight, but also with a series of local Pauli gates represented by the LIMs. This allows LIMDDs to represent all states at least as succinctly as QMDDs and stabilizer tableaus. Additionally, LIMDDs can efficiently represent states which cannot be represented efficiently with either the stabilizer formalism or QMDDs [337], for instance $|T\rangle \otimes |G\rangle$, where $|T\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)$ and $|G\rangle$ is a stabilizer state.

The interpretation of a LIMDD is similar to that of a QMDD. Each node still corresponds to a complex vector and, when following an edge, the vector given by the child node is still multiplied by the weight on the followed edge. For LIMDDs, rather than only multiplying the vector with a complex scalar, it is now additionally multiplied by the tensor product of the single-qubit gates on the incoming edge. This is illustrated in the following example.

Example 4.6. Consider again the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ from Example 4.1. Figure 4.2c shows the LIMDD of this state. Note that it uses one node less than the corresponding QMDD since it only requires one node q_0 due to the X operation annotated to the left out-edge of q_1. The node labelled q_0 represents the vector $|0\rangle$. From the root, following the left edge from node q_1 gives the vector $1/\sqrt{2} \cdot \mathbb{I}|0\rangle$, while following the right edge gives $1/\sqrt{2} \cdot X|0\rangle = 1/\sqrt{2} \cdot |1\rangle$. Note that correspondence is shown for illustration purposes only as working with decision diagrams does not require explicitly storing the vector at any point.

4.3.3 The need for a LIMDD implementation

As discussed, stabilizer states and Clifford circuits are ubiquitous in many quantum computing algorithms. Moreover, in the context of verifying quantum circuits, stabilizer states serve as good candidates for counterexamples. Therefore, we stand to profit twice from the exponential advantage that LIMDDs promise over existing decision diagrams: first, since stabilizers capture the symmetries present in all quantum states, LIMDDs likely improve universal simulation; second, when we want to verify a quantum circuit by reduction to simulation with a random stabilizer state, the Pauli-LIMDD is guaranteed to efficiently represent at least the initial state, whereas the QMDD is likely exponential [337, Appendix B].

However, the asymptotic advantage of LIMDDs comes with a price. They require both additional memory for bookkeeping the LIMs on edges and additional time for calculating canonical form of nodes. To the best of our knowledge, it is still unknown how this affects the memory and time use in practice because so far an implementation is absent. Existing implementations of decision diagrams in the classical domain [154, 183, 213, 304, 326] and the quantum domain [4, 242, 332, 337, 342, 369] have shown that translating the concept of a decision diagram into an efficient and usable program or library is far from trivial. LIMDDs are no exception to this rule and come with new challenges regarding the handling of the LIMs in the nodes and edges of the decision diagram.

4.4 Implementation of LIMDDs

As discussed in the previous sections, LIMDDs scale exponentially better in many cases compared to QMDDs. However, this advantage comes with an increased overhead to keep track of the local invertible maps annotated to nodes and edges in the decision diagram. Efficient management of this additional information is paramount to implement LIMDDs efficiently. Further crucial ingredients for efficient decision diagram implementations are canonicity and dynamic programming. Canonicity ensures that the diagram is never larger than necessary and uniquely represents a quantum state (in QMDDs) or Boolean function (in BDDs). Dynamic programming ensures that manipulation operations, such as gate applications or measurements, take polynomial time in the size of the diagram representing the state. To ensure canonicity, the implementation must put nodes in canonical form, as worked out in [337], and also store them in a corresponding table. To implement dynamic programming, LIMs must be normalized and stored in caches.

This section discusses both established techniques in developing implementations of decision diagrams in Sec. 4.4.1 as well as new approaches to efficiently handle the LIMs annotated to the edges of LIMDDs in Sec. 4.4.2. The established and correspondingly adopted techniques include dynamically-sized unique tables, garbage collection, compute tables, as well as indirect storing of complex numbers [369]. While the aforementioned techniques lay a solid foundation for the LIMDDs, they are not sufficient. Efficient approaches to store and manipulate the annotated local invertible maps are required to exploit the potential of LIMDDs.

4.4.1 Established techniques

Implementations of various types of (predominantly classical) decision diagrams have been proposed in the last decades [4,154,183,213,242,304,326,332,337,342]. Over this time, a lot of effort has been put into translating abstract concepts of decision diagrams into concrete instructions that run efficiently on classical computers. Multiple parts and "tricks" of these implementations are reusable for the LIMDD implementation as well. The following list provides a brief description of the most important building blocks:

Unique Tables store the nodes of the decision diagram and enable efficient detection of redundant nodes. These tables are commonly implemented as hash maps storing the nodes with two levels of indirection: the first level gives the qubit (or variable) and the second level is the hash value of the node, which is recursively calculated from the weights of the out-edges and hashes of the respective successors. The subsequent *strong canonical* identifier [59] of a node is the pointer into the memory of the unique tables, enabling access via constant-time de-referencing of the pointer.

When a new node is created, the unique tables are checked for already existing equivalent nodes. If an equivalent node exists, this node is re-used, otherwise the new node is stored in the unique tables.

Compute Tables cache results of operations to implement dynamic programming, avoiding repetitions of the same calculation. Intuitively, the more compact the decision diagram, the more paths (from root to leaf) traverse through the same nodes in the decision diagram. We can avoid processing exponentially many paths by hashing the operands of the recursive operation that traverses these paths in the diagram. The compute tables are implemented as individual hash tables for different operations, where the hash is calculated from the operand nodes.

Additionally, the compute tables are a key concept that enables efficient operations on decision diagrams by enabling dynamic programming. Without them, operations such as multiplication during circuit simulation would always be exponential, since no previous result could be re-used.

Handling of Complex Numbers requires special consideration to ensure that the limited accuracy of floating point numbers does not lead to wrong results. Two key aspects of these considerations are the introduction of a tolerance in the comparison of the components of the complex numbers and storing the components in a dedicated table to exploit the memory address as strong canonical form (with constant-time dereferencing).

Garbage Collection is frequently run to remove entries of the aforementioned tables that are not needed anymore, e.g., after each applied operation in quantum circuit simulation. For each node, a reference count is used to keep track of its state. Upon removal of nodes, garbage collection is also run on the other tables, such as the compute tables, so that no invalid pointers remain in memory—preventing an inconsistent state between the tables and subsequent illegal memory accesses.

The established techniques described above are used in existing packages of decision diagrams for quantum computing. However, LIMDDs require additional functionality to manage the LIMs which are annotated to edges and nodes. The next section describes the techniques employed to efficiently integrate the information on local invertible maps into the decision diagrams.

4.4.2 Implementing Local Invertible Maps

Efficient handling of the LIMs annotated to the nodes and edges in the LIMDDs is *the* requirement to actually realize the exponential advantage over QMDDs for certain quantum circuit simulations. Recall that a LIM consists of a complex factor and a Pauli operator $P = P_1 \otimes ... \otimes P_n$ with $P_i \in \{\mathbb{I}, X, Y, Z\}$ denoting a local Pauli operator. We call the latter a *Pauli string*.

The proposed LIMDD implementation still provides a strong canonical form for the nodes to ensure canonicity. We implement the canonical form presented in [337], with minor changes described below, which entails, among others, finding the lexicographically minimal Pauli strings for both of a node's outgoing edges, and possibly swapping the two children. The LIMs are stored in a new table to enable constant-time decisions whether two LIMs are equal. Additionally, because the all-identity operator occurs very frequently, we "hardcode" this as null pointer, to prevent many lookups to the LIM table.

In line with existing work [3], we represent a Pauli operator using two bits, so that the operators $\mathbb{I}, X, Y, Z$ are represented by $00, 01, 11, 10$, respectively, and a Pauli string of n operators is stored using $2n + 2$ bits, using 2 extra bits to store a scalar factor in $\{\pm 1, \pm i\}$. This enables efficient multiplication of Pauli operators, namely, the product of two LIMs is obtained by XORing their respective bit strings. For QMDDs, the diagram can be traversed by following edges, which is accomplished simply by dereferencing a pointer and multiplying the weight of the considered edge. For LIMDDs, following an edge is slightly more involved, since the local invertible map can affect each level downwards. To keep track of the LIMs to be applied and to avoid creating a new decision diagram for each followed edge, we keep auxillary information about the current LIM during each step of the traversal.

We now briefly list the biggest changes that are required to turn a QMDD package into a LIMDD package.

Putting nodes in canonical form ensures canonicity, which keeps the diagram as small as possible, by allowing nodes representing redundant subvectors to be merged. We use the canonicity scheme for LIMs as proposed in [337]. In this scheme, a node v always has the identity LIM $\mathbb{I}_2^{\otimes n}$ on its 0-edge and a LIM P on its 1-edge, such that P is the lexicographically minimal LIM possible, in the sense that using any smaller LIM results in a node v' which is not Pauli-isomorphic to v. Since the LIM P depends only on the state vector that the node represents, and is minimal in a precise way, this makes the node canonical. Consequently, the diagram will merge two nodes whenever they represent two Pauli-isomorphic subvectors. This minimal P is found by first finding the stabilizer tableaux (see Sec. 4.2.2) of v's two children states, which requires time $\mathcal{O}(n^3)$. This approach amortizes the cost of computing canonical LIMs over the entire DD structure; in other words, to construct a canonical LIM for node v, we only need to inspect the stored stabilizer generator sets of v's children (and not their descendants). This step, of constructing these groups, presents the biggest added computational overhead of all changes, namely the time required for making a new node increases from $\mathcal{O}(1)$ to $\mathcal{O}(n^3)$ because of the need to find stabilizer groups. Still, as shown in [337], this overhead enables a asymptotically exponential advantage.

Edge Weight Normalization is part of making the node canonical. We employ the normalization scheme from [158], which differs from the one proposed in [337]. Namely, when choosing the weights α_0, α_1 on the out-edges of a node, we require that $|\alpha_0|^2 + |\alpha_1|^2 = 1$, as opposed to requiring the left-most non-zero edge weight to be normalized to 1 (meaning that α_0 has to be either 0 or 1). This allows us to better take advantage of the existing cache for complex numbers and faster sampling from the decision diagram. Given such numbers α_0, α_1, we have a choice between α_1 and $-\alpha_1$, and we choose the one having nonnegative imaginary part; ties are broken by choosing the one with nonnegative real part. If we choose $-\alpha_1$, then we correct for this by multiplying the LIM on the incoming edge by $Z \otimes \mathbb{I}^{\otimes n-1}$.

The Operation Cache for LIMDDs allows us to improve the caching of the ADD operation results to potentially be more succinct and achieve more cache hits. Specifically, we get a cache hit on input $A\left|v\right\rangle + B\left|w\right\rangle$ whenever a previous call to ADD had input $C\left|v\right\rangle + D\left|w\right\rangle$ satisfying $A^{-1}B\left|w\right\rangle = C^{-1}D\left|w\right\rangle$. We implement this using the caching algorithm from [337], namely, on input $A\left|v\right\rangle + B\left|w\right\rangle$, if the result is edge $E\left|r\right\rangle$, then we add $\textsc{Cache}[F, v, w] := E\left|r\right\rangle$, where F is a canonically chosen LIM determined by A, B, and w.

The LIM Table stores the LIMs on the edges and the LIMs which generate a state's stabilizer group, so that common LIMs are shared in the LIMDD, thus reducing the total memory footprint. Multiple LIMDD sub-routines make use of a state's stabilizer group, e.g., for finding canonical edge labels. We choose to construct a set of LIMs which generate a state's stabilizer group as soon as that state's node is created, using the algorithm by [337], and we store these LIMs in the LIM Table. LIMs that are no longer required are identified via reference counting and removed during garbage collection.

We focus on the efficient storing and manipulation of LIMDDs for quantum states and continue to use QMDDs from the original package [369] to represent quantum operations as explained in Sec. 4.3.2, so that simulation can still be conducted in a fashion similar to [374]. This does not present a limitation of the approach, since QMDDs efficiently represent all gates considered in this work. Namely, we only use single-qubit gates with arbitrary controls, and these for gates the size of the diagram scales as $\mathcal{O}(n)$. Therefore, storage of the quantum state, rather than the matrix, remains the main memory bottleneck in common simulation scenarios.

To ensure the validity of the LIMDD implementation, we performed extensive tests on more than 1700 quantum circuits of varying sizes, parameters and complexity, ensuring that the intermediate state after every gate is the same as that found by the QMDD.

The next sections provides a case study that compares LIMDDs against QMDDs based on the Quantum Fourier Transform, which is an important building block of many quantum algorithms.

4.5 Case study

In this section, we provide the results obtained by an experimental case study of the implementation presented in this chapter. To this end, we created a complete, open-source LIMDD package in C++ available at `https://github.com/cda-tum/ddsim/tree/limdd`, based on an existing open-source implementation for QMDDs provided in [350, 369]. The motivation of the case study is an investigation on the extend that the theoretically proven advantage over QMDDs applies in an actual implementation.

In order to demonstrate the efficacy of the resulting implementation and thereby for the first time empirically comparing LIMDDs and QMDDs, we conducted quantum circuit simulation of a circuit which implements the Quantum Fourier Transform (QFT). The QFT is a common subroutine which is used by many quantum algorithms (notably order-finding in Shor's algorithm, phase estimation, and solving the hidden subgroup problem [240]); thus, verifying the correctness of this circuit is a useful step in the compilation toolchain of many quantum algorithms. We consider QFT circuits for various numbers of qubits, $n = 2 \dots 24$. We simulate the QFT on n qubits using a random n-qubit stabilizer state as the input. This stabilizer state is prepared by prepending a random Clifford circuit with $10 \cdot n$ gates, and then simulating from the initial state $|0\rangle^{\otimes n}$; the output of this circuit is a random stabilizer state.

The evaluation was conducted for QMDDs and LIMDDs on a server running GNU/Linux and GCC-10.3.0 with an AMD Ryzen 9 3950X running at 3.5 GHz and 128 GiB memory.

The results in Figure 4.3 show that LIMDDs outperform QMDDs on large instances (from $n = 19$ qubits and up), whereas QMDDs outperform LIMDDs on small instances (up to and including 18 qubits). This is most pronounced at 24 qubits, where LIMDDs are about five times faster. These results are expected: LIMDDs are proven to be asymptotically faster, but this comes at the price of adding a lot of computational

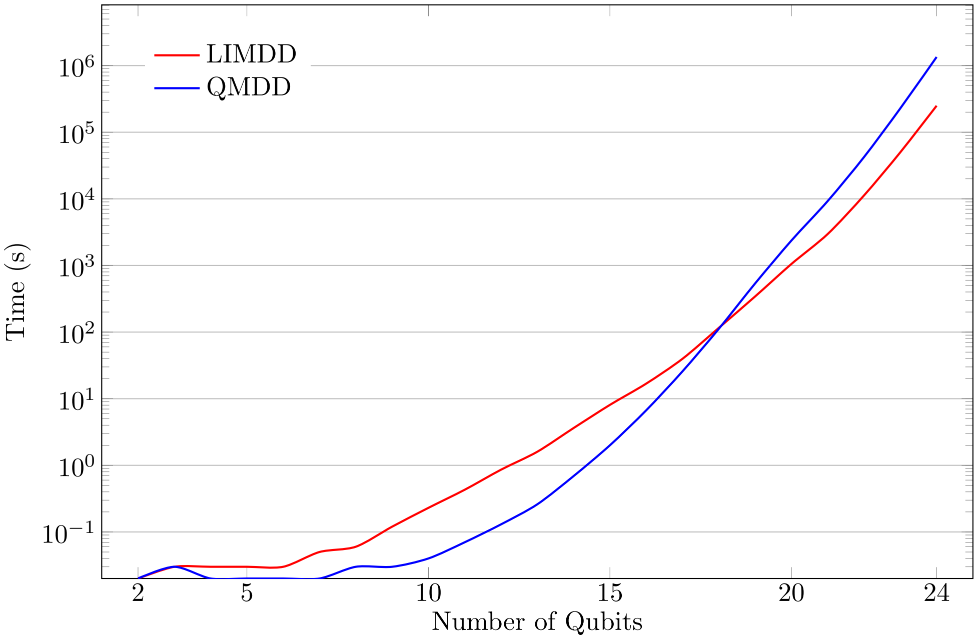


Figure 4.3: Quantum Fourier Transform Simulation on Random Stabilizer States

overhead in the handling of the LIMs on the edges, as explained in Section 4.4. The data show that this overhead pays off in the long run where the asymptotically better performance becomes realized in practice. In the graph, this translates into the fact that the LIMDD line is less steep than the QMDD line. The LIMDD is still small (it has $\mathcal{O}(n)$ nodes) when it finishes preparing a random stabilizer state and starts simulating the QFT, whereas the QMDD is already very large (it has $2^{\mathcal{O}(n)}$ nodes) at this point. At the end of the QFT, both types of decision diagrams are almost fully populated (i.e., almost of maximum size), since the state after the QFT does not possess much redundancy to be exploited. Generally, applying a gate to a small decision diagram is more efficient than applying the same gate to a large decision diagram, so especially the first few gates of the QFT can be applied quickly by LIMDDs, which eventually leads to the better runtime. In summary, while the additional overhead of LIMDDs outweights the lower complexity for small circuits, they can demonstrate their advantage as the circuit size increases.

4.6 Discussion

We pause to consider how effective this method is, and how effective future extensions may be expected to be. Kueng and Gross [190] showed that, given two unitaries U, V the *expected fidelity* $| \langle g| U^\dagger V |g\rangle |^2$ given a random stabilizer states $|g\rangle$ approaches the *average fidelity* of the matrix $U^\dagger V$. In this respect, therefore, random stabilizer states are as good as Haar random states. In other words, random stabilizer states are optimal in the black-box model, in the very narrow sense that they achieve optimal expected fidelity. This may be surprising, because Haar random states require an exponential amount of classical memory to represent, and require an exponential number of quantum gates to prepare, whereas stabilizer states require only $\mathcal{O}(n^2)$ memory to represent and $\mathcal{O}(n^2)$ gates to prepare.

However, there are several caveats to this optimality. First, the probability of finding a counterexample when using a random stabilizer state is typically less than when using a Haar state. That is, for given circuits U, V, it is possible that almost all stabilizer states $|g\rangle$ satisfy $\langle g| U^\dagger V |g\rangle = 1$ even when $\mathcal{F}_{\text{avg}}(U, V) < 1$. To alleviate this limitation, we might choose to use a different set of random input states, preferably a set which forms a complex projective t-design for large t (see [190]). However, such states are, by design, not structured, whereas computational methods are effective only in cases where the states have structure. We see that there is, therefore, a tension: we are looking for a set of states which is *random enough* to be a good t-design, but which is *structured enough* that computational methods are effective.

Second, in the above we present Haar-uniform random states as the ideal random set; but even these states have limitations. For example, even Haar-uniform random states do not effectively distinguish the identity gate from the multi-controlled Z gate (that is, the Z gate with $n-1$ control qubits and one target qubit). Namely, the average fidelity of these operators is $1 - \mathcal{O}(2^{-n})$, and in fact the fidelity of $U|g\rangle$ and $V|g\rangle$ is almost surely almost equal to this same expected value when $|g\rangle$ is a Haar-uniform random state. In other words, even in theory, the approach using random inputs has limitations. It is not yet clear how large this limitation is on practical circuits: future research into bugs on quantum computers (such as [166, 167, 251, 269]) may reveal whether, in practice, it is typical that a bug in an implementation U of a quantum circuit leads to a new (buggy) circuit V such that the average fidelity of U and V is very low. For now, we can say that it is not self-evident whether simulation on random input states will be an effective method for circuit equivalence in practice.

Lastly, we remark that testing using random input is a black-box technique, whereas simulation using decision diagrams is a white-box technique. Since we are in a white-box setting (i.e., we know all the gates) it is not obvious that a white-box technique (i.e., DDs) is a good match for a black-box methodology.

4.7 Conclusions

In this chapter, we presented the first implementation of *Local Invertible Map Decision Diagrams* (LIMDDs). The implementation includes techniques adapted from other decision diagram packages (both classical and quantum) that are tried and tested, as well as new considerations to efficiently handle *Local Invertible Maps* (LIMs). By this, we enable the potential of LIMDDs to be realized in practice. A case study confirm that LIMDDs provide an advantage for the classical simulation of quantum circuits that exceed a certain complexity, as shown by the Quantum Fourier Transform. The resulting open-source C++ implementation is available under the MIT license via `https://github.com/cda-tum/ddsim/tree/limdd`.

Chapter 5

A Quantum Knowledge Compilation Map

Quantum computing is finding promising applications in optimization, machine learning and physics, leading to the development of various models for representing quantum information. Because these representations are often studied in different contexts (many-body physics, machine learning, formal verification, simulation), little is known about fundamental trade-offs between their succinctness and the runtime of operations to update them. We therefore analytically investigate three widely-used quantum state representations: matrix product states (MPS), decision diagrams (DDs), and restricted Boltzmann machines (RBMs). We map the relative succinctness of these data structures and provide the complexity for relevant query and manipulation operations. Further, to chart the balance between succinctness and operation efficiency, we extend the concept of rapidity with support for non-canonical data structures, so that we can study several popular such data structures in this chapter. In particular, we show that MPS is at least as rapid as some DDs.

The aim of this chapter is to contribute to Research Question 2:

> **Research question 2.** *How can we analytically compare the relative strengths of data structures which represent quantum states?*

This chapter contributes to this question by articulating what we believe to be the most informative criteria on which such data structures should be compared: rapidity, succinctness, and tractability, *in that order*. It presents an exposition of such a comparison by systematically comparing several popular data structures on these criteria. In particular, we provide an easy-to-use framework for answering which of two data structures has better *rapidity*. By providing a knowledge compilation map for quantum state representation, this chapter contributes to our understanding of the inherent time and space efficiency trade-offs in this area.

5.1 Introduction

Quantum computing is showing great potential in various artificial intelligence endeavors, spanning tasks such as information retrieval [343], machine learning [234,358] and satisfiability [280]. In the other direction, AI is also crucial in quantum circuit compilation [57], quantum error correction [238,307] and quantum state learning [358]. Hence, we need good classical models for quantum information.

Various classical data structures have been proposed for representing quantum information, including decision diagrams (DDs; [227, 331]), tensor networks (TNs; [284]), matrix product states (MPS; [248]) also called tensor trains —a specialization of TNs— and restricted Boltzmann machines (RBMs; [104]) with complex values, a specialization of neural quantum states [78]. Recently, we have seen new applications for RBM [246], MPS [335], and also combinations of TNs and DDs [163], MPS and decision diagrams [72], TNs and probabilistic graphical models [133], and DDs with quantum circuits (LIMDDs; [337], introduced in Chapter 3).

However, fundamental differences between these data structures have not yet been studied in detail. This choice between different structures introduces an important trade-off between *size* and *speed*, i.e., how much space the data structure uses, versus how fast certain operations, such as measurement, can be performed. This trade-off plays a crucial role in other areas as well, and has already been illuminated for the domain of explainable AI [23] and logic [97,113], but not yet for quantum information.

In this chapter, we analytically compare for the first time several data structures for representing and manipulating quantum states, motivated by the following three applications. First, *simulating quantum circuits*, the building block of quantum com-

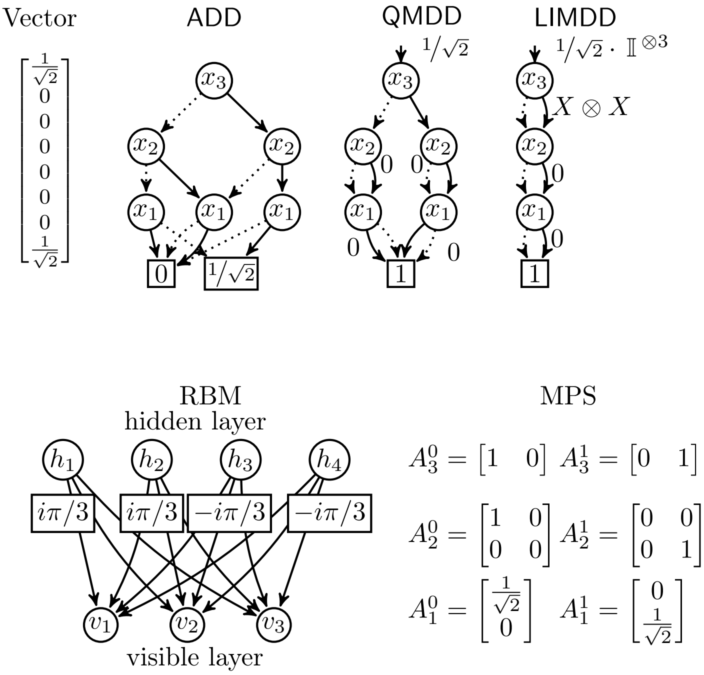

Vector
ADD
QMDD
LIMDD
$1/\sqrt{2}$
$1/\sqrt{2} \cdot \mathbb{I}^{\otimes 3}$
$X \otimes X$
x_3
x_2
x_1
0
1
$1/\sqrt{2}$
RBM
hidden layer
visible layer
h_1 h_2 h_3 h_4
$i\pi/3$ $i\pi/3$ $-i\pi/3$ $-i\pi/3$
v_1 v_2 v_3
MPS
$A_3^0 = \begin{bmatrix} 1 & 0 \end{bmatrix}$ $A_3^1 = \begin{bmatrix} 0 & 1 \end{bmatrix}$
$A_2^0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ $A_2^1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$
$A_1^0 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$ $A_1^1 = \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$

Figure 5.1: The 3-qubit GHZ state $1/\sqrt{2}(|000\rangle + |111\rangle)$, displayed using different data structures. The unlabelled edges for ADD, QMDD, LIMDD have resp. label 1, 1, $\mathbb{I}$. In the RBM, the weights of edges incident to h_1, h_2 (h_3, h_4) are all $i\pi/3$ ($-i\pi/3$); the hidden node biases $(\beta_{h_1}, \beta_{h_2}, \beta_{h_3}, \beta_{h_4}) = i\pi \cdot (1/3, 2/3, -1/3, -2/3)$; the visible node biases $\alpha_{v_1} = \alpha_{v_2} = \alpha_{v_3} = 0$.

putations, is a crucial tool for predicting the performance and scaling behavior of experimental devices with various error sources, thereby guiding hardware development [312, 374]. Next, *variational methods* are the core of *quantum machine learning* [37, 106] and solve quantum physics questions such as finding the lowest energy of a system of quantum particles [78, 122]. Finally, *verifying* if two quantum circuits are equivalent is crucial for checking if a (synthesized or optimized) quantum circuit satisfies its specification [18, 75, 127].

We focus on data structures developed for the above applications. In particular, we focus on: algebraic decision diagrams (ADD), semiring-labeled decision diagrams (QMDD, also called QMDD), local invertible map decision diagrams (LIMDD), matrix product states (MPS) and restricted Boltzmann machines (RBM). For those, we study *succinctness*, *tractability* and *rapidity*:

Succinctness & Tractability. In Section 5.3, we find that the succinctness of MPS, RBM and LIMDD are incomparable. We also find that MPS is strictly more succinct than QMDD, whereas previous research had suggested that these were incomparable [72]. In Section 5.4, we give states which QMDD and LIMDD can compactly represent, but which take exponential time to apply Hadamard and swap gates. Finally, we prove that computing fidelity and inner product in RBM and LIMDD is intractable assuming the exponential time hypothesis.
Rapidity. Considering tractability and succinctness separately can deceive; for instance, when representing a quantum state as an amplitude vector, most operations are polynomial time (i.e., tractable), but this is only because this vector has exponential size in the number of qubits. To mend this deficiency, [194] introduced the notion of *rapidity*, which reflects the idea that exponential operations may be preferable if a representation can be exponentially more succinct.

In Section 5.5, we generalize the definition of rapidity for non-canonical data structures, which allows us to study rapidity of MPS and RBM. We also give a simple sufficient condition for data structures to be more rapid than others (for all operations). We use it to settle several rapidity relations, showing surprisingly that MPS is strictly more rapid than QMDD. Since we are unaware of a previous comparison, our knowledge was consistent with them being incomparable.

5.2 Data structures for quantum states

This section gives the preliminaries that are necessary to understand this chapter. We present a more elaborate introduction to quantum computing in Section 2.2 and an introduction to decision diagrams in Section 2.3 – we note that those texts do not treat matrix product states or restricted Boltzmann machines.

5.2.1 Quantum information

A quantum state φ on n quantum bits or qubits, written in Dirac notation as $|\varphi\rangle$, can be represented as a vector of 2^n complex *amplitudes* $a_k \in \mathbb{C}$ such that the vector has unit norm, i.e. $\sum_{k=1}^{2^n} |a_k|^2 = 1$, where $|z| = \sqrt{a^2 + b^2}$ is the modulus of a complex number $z = a + b \cdot i$ with $a, b \in \mathbb{R}$. We also write $a_k = \langle k|\varphi\rangle$. Two quantum states $|\varphi\rangle, |\psi\rangle$ are equivalent if $|\varphi\rangle = \lambda |\psi\rangle$ for some complex λ. The inner product between two quantum states $|\varphi\rangle, |\psi\rangle$ is $\langle\varphi|\psi\rangle = \sum_{k=1}^{2^n} (\langle k|\varphi\rangle)^* \cdot \langle k|\psi\rangle$ where $z^* = a - bi$ is the complex conjugate of the complex number $z = a + b \cdot i$. The fidelity $|\langle\varphi|\psi\rangle|^2 \in [0, 1]$ is a measure of closeness, with fidelity equalling 1 if and only if $|\varphi\rangle$ and $|\psi\rangle$ are equivalent.

A quantum circuit consists of gates and measurements. An n-qubit *gate* is a $2^n \times 2^n$ unitary matrix which updates the n-qubit state vector by matrix-vector multiplication. A k-local gate is a gate which effectively only acts on a subregister of $k \leq n$ qubits. Often-used gates go by names Hadamard (H), T, Swap, and the Pauli gates X, Y, Z. Together with the two-qubit gate controlled-Z, the H and T gate form a universal gate set (i.e. any quantum gate can be arbitrarily well approximated by circuits of these gates only). Next, a (computational-basis) measurement is an operation which samples n bits from (a probability distribution defined by) the quantum state, *while also updating the state.*

We now elaborate on the application domains from Section 5.1. *Simulating a circuit* using the data structures in this chapter starts wlog by constructing a data structure for some (simple) initial state $|\varphi_0\rangle$, followed by manipulating the data structure by applying the gates and measurements $A, B, ...$ of the circuit one by one, i.e.: $|\varphi_1\rangle = A|\varphi_0\rangle$, $|\varphi_2\rangle = B|\varphi_1\rangle$, etc. The data structure supports strong (weak) simulation, if, for each measurement gate, it can produce (a sample of) the probability function (as a side result of the state update). Next, the quantum circuit of *variational methods* consist of a quantum circuit; then the output state $|\varphi\rangle$ is used to compute $\langle\varphi|O|\varphi\rangle$ for some linear operator O (an *observable*). This computation reduces to $\langle\varphi|\psi\rangle$ with $|\psi\rangle := O|\varphi\rangle$ (i.e., computing simulation and fidelity). Last, *circuit verification* relies on checking approximate or exact equivalence of quantum states. This extends to unitary matrices (gates), which all data structures from this chapter can represent also, but which we will not treat for simplicity.

5.2.2 Data structures

We now define the data structures for representing quantum states considered in this chapter, with examples in Figure 5.1.

Definition 5.1 (Inspired by [113]). A *quantum-state representation (language)* is a tuple $(D, n, |.\rangle, |.|)$ where D is a set of data structures. Given $\alpha \in D$, $|\alpha\rangle$ is the (possibly unnormalized) quantum state it represents (i.e., the interpretation of α), $|\alpha|$ is the size of the data structure, and $n(\alpha)$, or n in short, is the number of qubits of $|\alpha\rangle$.

Finally, each quantum state should be expressible in the language.

We will often refer to a representation as a data structure. We define $D^{\varphi} \triangleq \{\alpha \in D \mid |\alpha\rangle = |\varphi\rangle\}$, i.e., the set of all data structures in language D representing state $|\varphi\rangle$. In line with quantum information (see Sec. 5.2.1), we say that data structures α, β are *equivalent* if $|\alpha\rangle = \lambda\,|\beta\rangle$ for some $\lambda \in \mathbb{C}$.

Vector. A state vector α is a length-2^n array of complex entries $a_k \in \mathbb{C}^{2^n}$ satisfying $\sum_{k=1}^{2^n} |a_k|^2 = 1$, and interpretation $|\alpha\rangle = \sum_j \alpha_j\,|j\rangle$. Despite its size, the vector representation is used in many simulators [173]. We mainly include the vector representation to show that considering operation tractability alone can be deceiving.

Matrix Product States (MPS). An MPS M is a series of $2n$ matrices $A_k^x \in \mathbb{C}^{D_k \times D_{k-1}}$ where $x \in \{0,1\}$, $k \in [n]$ and D_k is the row dimension of the k-th matrix with $D_0 = D_n = 1$. The interpretation $|M\rangle$ is determined as $\langle \vec{y}|M\rangle = A_n^{x_n} \cdots A_2^{x_2} A_1^{x_1}$ for $\vec{y} \in \{0,1\}^n$. The size of M is the total number of matrix elements, i.e., $|M| = 2 \cdot \sum_{k=1}^{n} D_k \cdot D_{k-1}$. We will speak of $\max_{j \in [0\ldots n]} D_j$ as 'the' bond dimension.

Restricted Boltzmann Machine (RBM). An n-qubit RBM is a tuple $\mathcal{M} = (\vec{\alpha}, \vec{\beta}, W, m)$, where $\vec{\alpha} \in \mathbb{C}^n, \vec{\beta} \in \mathbb{C}^m$ for $m \in \mathbb{N}_+$ are *bias vectors* and $W \in \mathbb{C}^{n\times m}$ is a *weight matrix.* An RBM $\mathcal{M}$ represents the state $|\mathcal{M}\rangle$ as follows.

$$\langle \vec{x}|\mathcal{M}\rangle = e^{\vec{x}^T \cdot \vec{\alpha}} \cdot \prod_{j=1}^{m} (1 + e^{\beta_j + \vec{x}^T \cdot \vec{W}_j}) \tag{5.1}$$

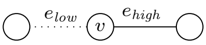
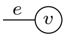
where $\vec{W}_j$ is the j-th column of W, β_j is the j-th entry of β and where we write $\vec{x}^T \cdot W_j$ to denote the inner product of the row vector $\vec{x}^T$ and the column vector $\vec{W}_j$ [80]. The size of $\mathcal{M}$ is $|\mathcal{M}| = n + m + n \cdot m$. We say this RBM has n *visible nodes* and m *hidden nodes.* A weight $W_{v,j}$ is an edge from the v-th visible node to the j-th hidden node. The j-th hidden node is said to contribute the multiplicative term $(1 + e^{\beta_j + \vec{x}^T \cdot \vec{W}_j})$.

Quantum Decision Diagrams (QDD). We define a Quantum Decision Diagram to represent a quantum state, based on the Valued Decision Diagram [113] as instantiated on a domain of binary variables (qu**bit**s) and a co-domain of complex values (amplitudes). A QDD α is a finite, rooted, directed acyclic graph (V, E), where each node v is labeled with an index $\mathsf{idx}(v) \in [n]$ and leaves have index 0. In addition, a 'root edge' e_R (without a source node) points to the root node. Each node has two outgoing edges, one labeled 0 (the low edge) and one labeled 1 (the high edge). In

Table 5.1: Various decision diagrams (DDs) treated by the literature. A DD represents a function $f\colon \{0,1\}^n \to R$ where the column Range specifies the set R. Here S is an arbitrary algebraic structure. The column *Merging strategy* lists the conditions under which two nodes v, w, representing subfunctions $f, g\colon \{0,1\}^k \to \mathbb{R}$ are merged. Here $z, a \in \mathbb{C}$ are complex constants, P_i are Pauli gates and $f + a$ denotes the function defined by $f(x) + a$ for all x. The DDs in **bold underlined text** are treated in depth in this thesis. See Chapter 7 for a definition of *variable tree*; see Gergov and Meinel [129] for a definition of FBDD type. (This table is identical to Table 2.2).

Architecture	**(Quantum) decision diagrams (and variants)**	Range	**Node merging strategy**
variable order	Decision Tree	(any)	(no merging)
	BDD, [67] ZDD, [229] TBDD, [328] CBDD, [68] KFBDD, [103] DSDBDD, [46, 264] CCDD, [195] Partitioned ROBDD, [237] Mod-2-OBDD, [130] ROBDD$[\wedge_i]_{\mathcal{C}}$, [194] CFLOBDD [293]	$\{0,1\}$	$f = g$
	MTBDD [87], **QuIDD** [331]	$\mathbb{C}$	$f = g$
	ADD [27]	S	$f = g$
	SLDD$_\times$ [114, 352], **QMDD** [227, 374], TDD, [163] WCFLOBDD [296]	$\mathbb{C}$	$f = z \cdot g$
	SLDD$_+$ [114], EVBDD [193]	$\mathbb{C}$	$f = g + a$
	AADD [281], FEVBDD [308]	$\mathbb{C}$	$f = z \cdot g + a$
	LIMDD [337]	$\mathbb{C}$	$f = zP_1 \otimes \ldots \otimes P_n \cdot g$
	DDMF	$U(2)$	$M = \mathbb{I} \otimes \cdots \otimes id \otimes U \cdot R$
variable tree	**SDD**, [96] ZSDD, [245] TSDD, [111] VS-SDD [235]	$\{0,1\}$	$f = g$
	PSDD [180]	$\mathbb{R}$	$f = g$
FBDD type	FBDD [129], Partitioned ROBDD [237]	$\{0,1\}$	$f = g$

addition, edge $e = vw$ pointing to a node w with index k has a label $label(e) \in \mathcal{E}_k$ for some edge label set $\mathcal{E}_k$; in this chapter, $\mathcal{E}_k$ is a group (for QMDD and LIMDD below with 0 added). Also, each leaf node v has a label $\mathsf{label}(v) \in \mathcal{L}$. The size of a QDD is $|\alpha| = |V| + |E| + \sum_{v \in V} |\mathsf{label}(v)| + \sum_{e \in E \cup \{e_R\}} |\mathsf{label}(e)|$. For simplicity, we require that no nodes are skipped, i.e. $\forall vw \in E\colon id[v] = id[w] + 1$.[*] The semantics are:

- A leaf node v represents the value $\mathsf{label}(v)$.
- A non-leaf node  represents $|v\rangle = |0\rangle \otimes |e_{low}\rangle + |1\rangle \otimes |e_{high}\rangle$.
- An edge  represents $|e\rangle = \mathsf{label}(e) \cdot |v\rangle$.

Consequently, any node v at *level* k, i.e., with $\mathsf{idx}(v) = k$, is k edges away from a leaf (along all paths) and therefore represents a k-qubit quantum state (or a complex vector).

In this chapter, we consider the following types of QDDs. We emphasize that an ADD can be seen as a special case of QMDD, which is a special case of LIMDD.

ADD: $\forall k\colon \mathcal{E}_k = \{1\}\,,\ \mathcal{L} = \mathbb{C}$ [27].

QMDD: $\forall k\colon \mathcal{E}_k = \mathbb{C},\ \mathcal{L} = \{1\}$ [352].

LIMDD: $\forall k\colon\ \mathcal{E}_k = \textsc{PauliLIM}_k \cup \{0\},\ \mathcal{L} = \{1\}$, where $\textsc{PauliLIM}_k \triangleq \{\lambda P \mid \lambda \in \mathbb{C} \setminus \{0\}, P \in \{\mathbb{I}, X, Y, Z\}^{\otimes k}\}$, i.e., the group generated by k-fold tensor products of the single-qubit Pauli matrices [337]:

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

Two isomorphic QDD nodes (Definition 5.2) v, w with $|w\rangle = \ell\,|v\rangle$ can be *merged* by removing w and rerouting all edges $uw \in E$ incoming to w to v, updating their edge labels accordingly (i.e., $\mathsf{label}(uv) := \mathsf{label}(uw) \cdot \ell$). Table 5.1 summarizes merging strategy for the above QDDs and related versions. If all isomorphic nodes are merged, we call a QDD *reduced*. We may assume a QDD is reduced (and even canonical), since it can be reduced in polynomial time and manipulation algorithms keep the QDD reduced using a MakeNode operation [67, 114, 227, 337].

---

[*] Asymptotic analysis is not affected by disallowing node skipping as it yields linear-size reductions at best [183].

Definition 5.2 (Isomorphic nodes)**.** QDD node v is isomorphic to node w, if there exists an edge label $\ell \in \mathcal{E}_{\mathsf{idx}(v)}, \ell \neq 0$ such that $\ell \cdot |v\rangle = |w\rangle$. Since $\mathcal{E}_{\mathsf{idx}(v)}$ is a group, $\ell \in \mathcal{E}_{\mathsf{idx}(v)}$ implies $\ell^{-1} \in \mathcal{E}_{\mathsf{idx}(v)}$ so $\ell^{-1}|w\rangle = |v\rangle$. Hence isomorphism is an equivalence relation.

It is well known that the variable order greatly influences QDD sizes [56, 58, 96]. Our results here assume any variable order: For instance, when we say that some structure is strictly more succinct than a QDD, it means that there is no variable order for which the QDD is more succinct than the other structure (for representing a certain worst-case state).

5.3 Succinctness of quantum representations

In this section, we compare the succinctness of the data structures introduced in Section 5.2. A language L_1 is as succinct as L_2 if L_2 uses at least as much space to represent any quantum state as L_1, up to polynomial factors. Thus, a more succinct diagram is more expressive when we constrain ourselves to polynomial (or asymptotically less) space. We define a language L_1 to be *at least as succinct* as L_2 (written $L_1 \preceq_s L_2$), if there exists a polynomial p such that for all $\beta \in L_2$, there exists $\alpha \in L_1$ such that $|\alpha\rangle = |\beta\rangle$ and $|\alpha| \leq p(|\beta|)$. We say that L_1 is more succinct ($L_1 \prec_s L_2$) if

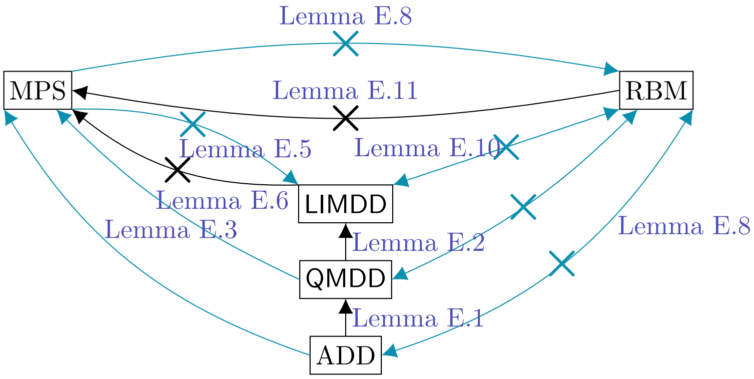


Figure 5.2: Succinctness relations between various classical data structures for representing quantum states. Solid arrows $A \to B$ denote $B \prec_s A$, i.e., B is strictly more succinct than A. Crossed arrows $A \not\twoheadrightarrow B$ denote a separation $B \not\preceq_s A$; a bidirectional crossed arrow implies incomparability. Blue arrows indicate novel relations that we identified.

L_1 is at least as succinct as L_2 but not vice versa.

The results of this section are summarized by Theorem 5.1 (Figure 5.2) and proved in App. E. We now highlight our novel results. First, we show that MPS $\prec_s$ QMDD, by (i) describing a simple and efficient way to write a QMDD as an MPS, and (ii) finding a family of states, which we call $|\mathrm{Sum}\rangle$, which have small MPS but which require exponentially large QMDD. It follows that MPS $\prec_s$ ADD. Second, we strengthen (ii) to show that the same state also requires exponentially large LIMDDs, thus we establish LIMDD $\not\preceq_s$ MPS. The reverse here also holds because MPS cannot efficiently represent certain stabilizer states, while LIMDD is small for all stabilizer states, as shown in [337]. Lastly, we show that RBMs can efficiently represent $|\mathrm{Sum}\rangle$. Since it is well-known that RBM explodes for parity functions, which are small for QDD [214], we establish incomparability of RBM with all three QDDs.

Theorem 5.1. The succinctness results in Figure 5.2 hold.

5.4 Tractability of quantum operations

In this section, we investigate for each data structure (DS) the tractability of the main relevant queries and manipulation operations for the different applications discussed Section 5.1.

By a *manipulation operation*, we mean a map $D^c \to D$ and by a *query operation* a map $D^c \to \mathbb{C}$, where D is a class of data structures and $c \in \mathbb{N}_{\geq 1}$ the number of operands. We say that a class of data structures D *supports* a (query or manipulation) operation $OP(D)$, if there exists an algorithm implementing OP whose runtime is polynomial in the size of the operands, i.e., $|\varphi_1| + \ldots + |\varphi_c|$.

The operations whose tractability we investigate are:

- **Sample**: Given a DS representing $|\varphi\rangle$, sample the measurement outcomes $\vec{x} \in \{0,1\}^n$ from measuring all qubits of $|\varphi\rangle$ in the computational basis.
- **Measure**: Given a DS representing $|\varphi\rangle$ and $\vec{x} \in \{0,1\}^n$, compute the probability of obtaining $\vec{x}$ when measuring $|\varphi\rangle$.
- **Gates** (**Hadamard**, Pauli **X,Y,Z**, Controlled-Z **CZ**, **Swap**, **T**): Given a DS representing $|\varphi\rangle$ and a one- or two-qubit gate U, construct a DS representing $U|\varphi\rangle$. This gate set is universal [63] and is used by many algorithms [329].

Table 5.2: Tractability of queries and manipulations on the data structures analyzed in this chapter (single application of the operation). A ✓ means the data structure supports the operation in polytime, a ✓' means supported in randomized polytime, and ✖ means the data structure does not support the operation in polytime. A ○ means the operation is not supported in polytime unless $P = NP$. ? means unknown. The table only considers deterministic algorithms (for some ? a probabilistic algorithm exists, e.g., for **InnerProd** on RBM). Novel results are blue and underlined.

		Queries					Manipulation operations						
		Sample	**Measure**	**Equal**	**InnerProd**	**Fidelity**	**Addition**	**Hadamard**	**X,Y,Z**	**CZ**	**Swap**	**Local**	**T-gate**
Vector		✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ADD	Section F.1	✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
QMDD	Section F.2	✓'	✓	✓	✓	✓	✖	✖	✓	✓	✖	✖	✓
LIMDD	Section F.3	✓'	✓	✓	○	○	✖	✖	✓	✓	✖	✖	✓
MPS	Section F.4	✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RBM	Section F.5	✓'	?	?	○	○	?	?	✓	✓	✓	?	✓

- **Local** (Local gates) as general case of Gates: Given a DS representing $|\varphi\rangle$, a constant $k \in \mathbb{N}_{\geq 1}$ and a k-local gate U, construct a DS representing the state $U \cdot |\varphi\rangle$.
- **Addition**: Given DSs for states $|\varphi\rangle, |\psi\rangle$, construct a DS representing the state $|\varphi\rangle + |\psi\rangle$.
- **InnerProd** (inner product) and **Fidelity**: Given DSs for states $|\psi\rangle, |\psi\rangle$, compute $\langle\varphi|\psi\rangle$ and $|\langle\varphi|\psi\rangle|^2$.
- **Equal** (Equality): Decide whether the states represented by two given data structures are equivalent.

We are motivated to study these operations by their applicability in the three application domains from Section 5.1. First, classical simulation of quantum circuits includes **Gates**, as well as **Measure** (strong simulation) and **Sample** (weak simulation). Although **Addition** is not, technically speaking, a quantum operation, we include it because addition of quantum states can happen due to quantum operations. For example, if we apply first a Hadamard gate, and then a measurement, to the state $|0\rangle|\varphi\rangle + |1\rangle|\psi\rangle$, we may obtain the state $|0\rangle \otimes (|\varphi\rangle + |\psi\rangle)$. For a data structure, this resulting state is at least as difficult to represent as the sum $|\varphi\rangle + |\psi\rangle$. Therefore, it is instructive to check which data structures support **Addition**: it allows us to explain why certain gates are not tractable for a data structure (such as the swap and general local gates). In particular, all decision diagram implementations support an explicit addition subroutine [86, 227]. In addition, **InnerProd** and **Fidelity** (and **Local**) are required in variational methods and quantum circuit verification involves **Eq**.

For the operations listed above and the languages from Section 5.2, we present an overview of existing and novel tractability results in Theorem 5.2 (Table 5.2), with proofs for all entries in App. F. The novel results in this work are the hardness results (denoted ○) for **InnerProd** and **Fidelity** on LIMDD and RBM (Theorem 5.3 and for **InnerProd** we can reduce from **Fidelity**) and unsupported manipulations (denoted ✖) on QMDD and LIMDD. More generally, our proof shows that computing **Fidelity** is hard for any data structure which succinctly represents both all graph states and the Dicke state, such as RBM and LIMDD. A fidelity algorithm for QMDD was mentioned by [71], but, to the best of our knowledge, had not previously been described or analyzed.

Theorem 5.2. The tractability results in Table 5.2 hold.

Theorem 5.3. Assuming the exponential time hypothesis, the fidelity of two states represented as LIMDDs or RBMs cannot be computed in polynomial time. The proof uses a reduction from the #EVEN SUBGRAPHS problem [169].

5.5 Rapidity of data structures

The tractability criterion, studied in Section 5.4, sometimes gives a skewed picture of efficiency. For example, looking naively at Table 5.2, it seems that ADD is faster than QMDD when applying a Hadamard gate. Yet there is no state for which applying the Hadamard gate on its QMDD representation is slower than on its ADD representation. So succinctness actually consistently mitigates the worst-case runtime behavior. To remedy this shortcoming, [194] introduced the notion of *rapidity* for canonical data structures.

In Definition 5.3, we generalize rapidity to support non-canonical data structures, such as MPS, RBM, d-DNNF [95] and CCDD [195]. To achieve this, Definition 5.3 requires that for a fixed input φ to ALG_1, among all equivalent inputs to ALG_2, there is one on which ALG_2 is at least as fast as ALG_1. It may seem reasonable to require, instead, that ALG_2 is at least as fast as ALG_1 on *all* equivalent inputs; however, in general, there may be no upper bound on the size of the (infinitely many) such inputs; thus, such a requirement would always be vacuously false.

In the following, we write $\mathit{time}(A, x)$ for the runtime of algorithm A on input x.

Definition 5.3 (Rapidity for non-canonical data structures)**.** Let D_1, D_2 be two data structures and consider some c-ary operation OP on these data structures. In the below, ALG_1 (ALG_2) is an algorithm implementing OP for D_1 (D_2).

(a) We say that ALG_1 is *at most as rapid as* ALG_2 iff there exists a polynomial p such that for each input $\varphi = (\varphi_1, \ldots, \varphi_c)$ there exists an equivalent input $\psi = (\psi_1, \ldots, \psi_c)$, i.e., with $|\varphi_j\rangle = |\psi_j\rangle$ for $j = 1 \ldots c$, for which $\mathit{time}(ALG_2, \psi) \leq p\,(\mathit{time}(ALG_1, \varphi))$. We say that ALG_2 is *at least as rapid as* ALG_1.

(b) We say that $OP(D_1)$ is *at most as rapid as* $OP(D_2)$ if for each algorithm ALG_1 performing $OP(D_1)$, there is an algorithm ALG_2 performing $OP(D_2)$ such that ALG_1 is at most as rapid as ALG_2.

We remark that, when applied to canonical data structures, Definition 5.3 reduces to the definition by Lai et al. (Lemma G.1), except that Lai et al. allow the input to be fully read by the algorithm: $\mathit{time}(ALG_1, x_1) \leq \mathsf{poly}(\mathit{time}(ALG_2, x_2)\underline{+|x_2|})$ (difference underlined). We omit this to achieve transitivity. Rapidity indeed has the desirable property that it is a preorder, i.e., it is reflexive and transitive, as Theorem 5.4 shows. App. G provides a proof.

Theorem 5.4. Rapidity is a preorder over data structures.

5.5.1 A sufficient condition for Rapidity

We now introduce a simple sufficient condition for rapidity in Theorem 5.5, allowing researchers to easily establish that one data structure is more rapid than another *for many relevant operations simultaneously.* Previously, such proofs were done for each operation individually [194]. We use this sufficient condition to establish rapidity relations between many of the data structures studied in this work. By a *transformation* f from data structure D_1 to D_2 we mean a map such that $|f(x_1)\rangle = |x_1\rangle$ for all $x_1 \in D_1$. We also need the notions of a weakly minimizing transformation (Definition 5.4) and a runtime monotonic algorithm (Definition 5.5).

Definition 5.4 (Weakly minimizing transformation)**.** Let D_1, D_2 be data structures. A transformation $f : D_1 \to D_2$ is *weakly minimizing* if f always outputs an instance which is polynomially close to minimum-size, i.e., there exists a polynomial p such that for all $x_1 \in D_1, x_2 \in D_2$ with $|x_1\rangle = |x_2\rangle$, we have $|f(x_1)| \leq p(|x_2|)$.

Definition 5.5 (Runtime monotonic algorithm)**.** An algortihm ALG implementing some operation on data structure D is *runtime monotonic* if for each polynomial s there is a polynomial t such that for each state $|\varphi\rangle$ and each $x, y \in D^{\varphi}$, if $|x| \leq s(|y|)$, then $\mathit{time}(ALG, x) \leq t(\mathit{time}(ALG, y))$.

Theorem 5.5 (A sufficient condition for rapidity)**.** Let D_1, D_2 be data structures with $D_1 \preceq_s D_2$ and OP a c-ary operation. Suppose that,

A1 $OP(D_2)$ requires time $\Omega(m)$ where m is the sum of the sizes of the operands; and

A2 for each algorithm ALG implementing $OP(D_2)$, there is a runtime monotonic algorithm ALG^{rm}, implementing the same operation $OP(D_2)$, which is at least as rapid as ALG; and

A3 there exists a transformation from D_1 to D_2 which is (i) weakly minimizing and (ii) runs in time polynomial in the output size (i.e, in time $\mathsf{poly}(|\psi|)$ for transformation output $\psi \in D_2$); and

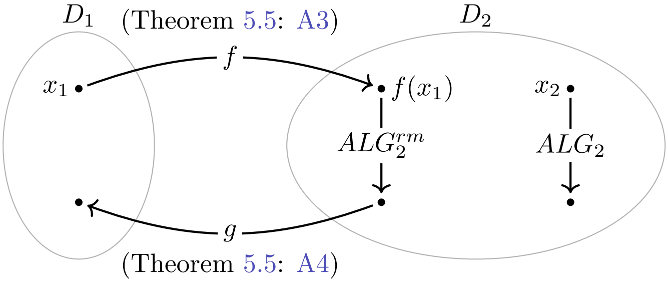


Figure 5.3: Visualization of the proof of Theorem 5.5 in case OP is a transformation operation: Given runtime monotonic algorithm ALG_2^{rm} implementing OP on language D_2, the composed algorithm $ALG_1 \triangleq g \circ ALG_2^{rm} \circ f$ for $OP(D_1)$ is at least as rapid as ALG_2. To prove this, we consider $x_2 \in D_2$ and an equivalent and at most only polynomially larger than $x_1 \in D_1$ and show that ALG_1 takes at most polynomially more time on x_1 than ALG_2 on x_2. ALG_1 is also runtime monotonic. Horizontally-aligned instances of data structures are equivalent, i.e. represent the same quantum state. App. G provides a proof.

A4 if OP is a manipulation operation (as opposed to a query), then there also exists a polynomial time transformation from D_2 to D_1 (polynomial time in the input size, i.e, in $|\rho|$ for transformation input $\rho \in D_2$).

Then D_1 is at least as rapid as D_2 for operation OP.

Figure 5.3 provides a proof outline that illustrates the need for (polynomial-time) transformations in order to execute operations on states represented in data structure D_1 on their D_2 counterpart. The operation on $f(x_1)$ is at most polynomially slower than on its counterpart x_2 because f produces a small instance (because f is weakly minimizing), and because ALG_2^{rm} is not much slower on such instances (because it is runtime monotonic). We opted for weakly minimizing transformations (rather than *strictly* minimizing transformations), because a minimum structure might be hard to compute and is not needed in the proof. Runtime monotonic algorithms are ubiquitous, for instance, most operations on MPS scale polynomially in the bond dimension and number of qubits [335]. Finally, we emphasize that for canonical data structures D, each algorithm is runtime monotonic and any transformation $D_1 \to D$ is weakly minimizing.

5.5.2 Rapidity between quantum representations

We now capitalize on the sufficient condition of Theorem 5.5 by revealing the rapidity relations between data structures for all operations satisfying A1 and A2. Theorem 5.6 shows our findings. We highlight the result that MPS is at least as rapid as QMDD in Theorem 5.7. Its proof provides the required transformations from MPS to QMDD and back.

Theorem 5.6. The rapidity relations in Figure 5.4 hold.

Theorem 5.7. MPS is at least as rapid as QMDD for all operations satisfying A1 and A2.

Proof sketch. Since QMDD is canonical, the runtime monotoncity (A2 of Theorem 5.5, see Definition 5.5) and weakly-minimizing (A3) requirements are satisfied automatically. Hence we only need to provide efficient transformations in both directions (A3-A4): We transform QMDD to MPS by choosing its matrices A_k^x to be the weighted bipartite adjacency matrices of the x-edges (low / high edges) between level k and $k-1$ nodes of the QMDD. In the other direction, an MPS can be efficiently transformed into an QMDD by contracting the first open index with $|0\rangle$ and then $|1\rangle$ to find the coefficients in the $|0\rangle$ and $|1\rangle$ parts of the state's Shannon decomposition; and repeating this recursively for all possible partial assignments. Dynamic programming using the fidelity operation (efficient for MPS) ensures that only a polynomial number of recursive calls are made. □

The relations not involving MPS involve QDDs. QDDs are canonical data structures as explained in Section 5.2, so that runtime monotonicity and weak minimization of transformations are automatically ensured. In Section G.5, we give detailed transformations between QDDs. Transformations between different QDDs can be realized using the well-known MakeNode procedure (see Section 5.2 and Section 2.3), in linear time in the resulting QDD size (using dynamic programming).

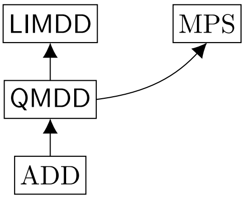


Figure 5.4: Rapidity relations between data structures considered here. A solid arrow $D_1 \to D_2$ means D_2 is at least as rapid as D_1 for all operations satisfying A1 and A2 of Theorem 5.5.

5.6 Related work

Darwiche and Marquis [97] pioneered the knowledge compilation map approach followed here. Fargier et al. [113] employ the same method to compare decision diagrams for real-valued functions. In order to mend a deficiency in the notion of tractability, Lai et al. [194] contributed the notion of *rapidity*, which we extend in Section 5.5

Alternative ways to deal with non-canonical data structures includes finding a canonical variant, as has been done for MPS [262], although it is not canonical for all states. For other structures, such as RBM, d-DNNF and CCDD (mentioned before) such canonical versions do not exist as far as we know, and might not be tractable.

Often the differences between DD variants in Table 2.2 are subtle differences in the constraints used for obtaining canonicity. While perhaps irrelevant for the (asymptotic) analysis, we emphasize that these definitions have great practical relevance. For instance, canonicity makes dynamic programming efficient, which enables fast implementations of QDD operations, while it also reduces the diagram size. Moreover, in practice the behavior of floating point calculations interacts with the canonicity constraints [369], significantly affecting performance. Like Fargier et al. [113], we do not analyze numerical stability and focus on asymptotic behavior.

The affine algebraic decision diagram (AADD), introduced by Sanner and McAllister [281], augments the QMDD as shown in Table 2.2. Their work proves that the worst-case time and space performance of AADD is within a multiplicative factor of that of ADD. The concept of rapidity makes explicit that this is the case only when equivalent inputs are considered for both structures. We note that AADD would be able to represent $|\mathrm{Sum}\rangle$ (a hard case for all QDDs studied here), so its succinctness relation with respect to RBM is still open. We omit Affine ADD (AADD) [281], since to the best of our knowledge these have not been applied to quantum computing yet. The LIMDD data structure is implemented and compared against QMDD in Chapter 4 (which is based on [338]).

Burgholzer et al. [72] compare tensor networks, including MPS, to decision diagrams, on slightly different criteria, such as abstraction level and ease of distributing the computational workload on a supercomputer. Hong et al. introduce the Tensor Decision Diagram, which extends the QMDD so that it is able to represent tensors [163] and their contraction. Context-Free-Language Ordered Binary Decision Diagrams (CFLOBDDs) [293, 295, 296] achieve a similar goal by extending BDDs with insights from visibly pushdown automata [10].

The stabilizer formalism is a tractable but non-universal method to represent and manipulate quantum states [3]. Its universal counterpart is the stabilizer decomposition-based method, introduced by Bravyi et al. [64], which relies on a linear combination of representations in the stabilizer formalism that is exponential only in the number of T gates in the circuit. Although this is a highly versatile technique [62], no super-polynomial lower bounds on their size are known at the moment, making it less useful to include them in a knowledge compilation map.

Similarly, we expect the relation between tree tensor networks [248] and SDDs [96] to be similar the relation between QMDD and MPS, since SDD also extend the linear

QDD variable order with a 'variable tree.'

Ablayev et al. [5] introduce the Quantum Branching Program, a branching program whose state is a superposition of the nodes on a given level, some of which are labeled *accepting*. An input string $x \in \{0,1\}^n$ determines how the superposition evolves. A string $x \in \{0,1\}^n$ is said to be accepted by the automaton if, after evolving according to x, a measurement causes the superposition to collapse to an accepting state with probability greater than $\frac{1}{2}$. Thus, the automaton accepts a finite language $L \subseteq \{0,1\}^n$. Since this diagram accepts a language, rather than represents and manipulates a quantum state, we cannot properly fit it into the present knowledge compilation map.

5.7 Discussion

We have compared several classical data structures for representing quantum information on their succinctness, tractability of relevant operations and rapidity. We catalogued all relevant operations required to implement simulation of quantum circuits, variational quantum algorithms (e.g., quantum machine learning) and quantum circuit verification. We have followed the approach of Darwiche and Marquis [97] and Fargier et al. [113] to map the succinctness and tractability of the data structures. In order to mend a deficiency in the notion of tractability that was noticed by several researchers, we additionally adopt and develop the framework of *rapidity*. We contribute a general-purpose method by which data structures, whether canonical or not, may be analytically compared on their rapidity.

Common knowledge says that there is a trade-off between the succinctness, and the speed of a data structure. In contrast, we find that, the more succinct data structures are often also more rapid. For example, we find that algebraic decision diagrams, QMDD, and matrix product states are each successively more succinct and more rapid. However, in practice, the QMDD has achieved striking performance on realistic benchmarks [374] due to a successful sustained effort to optimize the software implementation. But we emphasize that e.g. MPS was not developed with the intention for circuit simulation but for quantum simulation, and QMDD is vice versa. Therefore, more research is needed to compare the relative performance of these data structures on the various application domains in practice.

In future work, we could also consider unbounded operations (multiple applications of the same gate) and other data structures, e.g., tree tensor networks [248] and affine algebraic decision diagrams [281].

Chapter 6

The Power of Disjoint Support Decompositions in BDDs

The relative succinctness and ease of manipulation of different languages to express Boolean constraints is studied in knowledge compilation, and impacts areas including formal verification and circuit design. We give the first analysis of Disjoint Support Decomposition Binary Decision Diagrams (DSDBDD), introduced by Bertacco, which achieves a more succinct representation than Binary Decision Diagrams (BDDs) by exploiting Ashenhurst Decompositions. Our main result is that DSDBDDs can be exponentially smaller than BDDs.

This chapter contributes to Research Question 3:

> **Research question 3.** *Which classical decision diagrams might be effective for the analysis of quantum algorithms, if they were suitably adapted?*

Our answer is that we find the results of this chapter encouraging: the (classical) DSDBDD provides an example of a minor and conceptually simple modification of a DD which nevertheless leads to exponential improvements. Can such lessons inspire us to design a "quantum DSDBDD," which effects exponential improvement, too? Indeed, we formulate specific ways to draw inspiration from the DSDBDD to design such a "quantum DSDBDD" in Section 8.3. In the same chapter, we speculate about several other opportunities to improve quantum decision diagrams.

6.1 Introduction

Decision Diagrams are data structures for the representation and manipulation of Boolean functions. They are used for probabilistic reasoning [28, 91], verification [162, 221, 339], circuit design [300, 301, 347, 372] and simulation of quantum computing [242,333,337]. Since Bryant [67] popularized the Binary Decision Diagram (BDD), there has been a proliferation of different decision diagrams which use different architectures, e.g., ZDD [229], TBDD [328], CBDD [68], SDD [96], uSDD [322], FBDD [129]. Darwiche and Marquis [97] analytically compare the succinctness and tractability of manipulation operations (e.g., computing the logical OR of two functions) of these different diagrams and other representations, such as CNF, resulting in a *knowledge compilation map*. In particular, they elucidate the inherent tradeoff between succinctness and tractability: Some diagrams can be exponentially more succinct, but do not admit efficient manipulation and/or query operations, or vice versa (e.g., d-DNNF [97] strictly contains DDs and allows model counting in polynomial time, but no efficient algorithm for computing the logical OR is known; and SDDs can be exponentially more succinct than BDDs [58]).

The Disjoint Support Decomposition BDD [46, 264] (DSDBDD[*]) augments a BDD with disjunctive decompositions (sometimes called Ashenhurst-Curtis decompositions [8, 22]). They are canonical like BDDs and support the same queries and operations as BDDs (model counting, conjunction, negation, etc.). DSDBDDs have so far been deployed in only few applications, mostly in circuit verification [265]. In order to know whether efforts to deploy them elsewhere are likely to be fruitful, we make an initial step towards placing DSDBDDs on the knowledge compilation map. Our main result is that DSDBDDs can be exponentially smaller than BDDs. To this end, we give a function that yields the separation, drawing on the theory of expander graphs to show that its BDD cannot be small. As corollaries, we also clarify the relation between other languages. Finally, we also point out some open questions.

---

[*] No name has been given to this diagram, so we use DSDBDD in accordance with conventions in the literature.

6.2 Background and related work

A decision diagram is a data structure used to represent and manipulate Boolean functions. For an accessible exposition of decision diagrams, the interested reader may consult Section 2.3. For the purposes of this chapter, it suffices to give the following definition of a Binary Decision Diagram (BDD):

Definition 6.1 (Binary Decision Diagram (BDD))**.** A BDD is a rooted, directed acyclic graph. It has two leaves, labeled TRUE (or 1) and FALSE (or 0). A non-leaf node is called a *Shannon node*; it is labeled with (the index of) a variable and has two outgoing edges, called the *low edge* and the *high edge*. Each node v represents a Boolean function $[\![v]\!]$, defined inductively as follows. In the base case, the TRUE and FALSE leaves represent the constant functions $[\![\text{TRUE}]\!] = 1$ and $[\![\text{FALSE}]\!] = 0$,

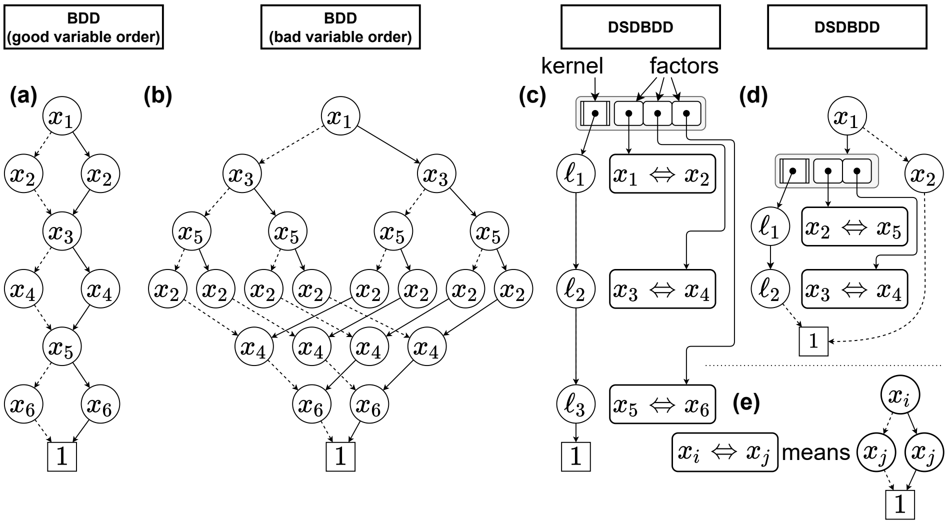

BDD (good variable order)
BDD (bad variable order)
DSDBDD
DSDBDD
kernel
factors
(a)
(b)
(c)
(d)
(e)
x_1
x_2
x_3
x_4
x_5
x_6
ℓ_1
ℓ_2
ℓ_3
$x_1 \Leftrightarrow x_2$
$x_3 \Leftrightarrow x_4$
$x_5 \Leftrightarrow x_6$
$x_2 \Leftrightarrow x_5$
x_i
x_j
$x_i \Leftrightarrow x_j$ means
1

Figure 6.1: (a) and (b): BDDs for the function $f \triangleq (x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4) \wedge (x_5 \Leftrightarrow x_6)$. Low (high) edges are drawn as dotted (solid) arcs. The FALSE Leaf and arcs which go to the FALSE Leaf are not drawn. These two BDDs use different variable orders: (a) uses $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$, whereas (b) uses $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$, which is why (b) is much bigger than (a). (c): a DSDBDD for the same function. The root node is a decomposition node, whose kernel and factors are indicated. For sake of clarity and compactness, in several parts of the figure we have “collapsed” the small BDD representing $x_i \Leftrightarrow x_j$ by drawing it as a single rectangle (e). (d): a DSDBDD which represents the function $f = \neg x_1 \wedge \neg x_2 \vee x_1 \wedge ((x_2 \Leftrightarrow x_5) \wedge (x_3 \oplus x_4))$ and whose root node is not a Decomposition node.

respectively. If a Shannon node v is labeled with variable x and has low edge to v_0 and high edge to v_1, then it represents the function $[\![v]\!] \triangleq \neg x \wedge [\![v_0]\!] \vee x \wedge [\![v_1]\!]$. A BDD is *ordered* if, on each path from the root to a leaf, each variable appears at most once and always in the same order. Two nodes u, v are called *equivalent* if they represent the same function, $[\![u]\!] = [\![v]\!]$. A BDD is *reduced* if there are no equivalent nodes. ◇

Figures 6.1(a) and (b) show examples of BDDs. These two BDDs represent the same function, $f \triangleq (x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4) \wedge (x_5 \Leftrightarrow x_6)$. They have a different shape, because they employ different variables orders, $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$ and $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$, respectively. In fact, if we generalized the functions and variable orders from $n = 6$ to $n > 6$, the corresponding BDD would stay linearly sized in the first case, but in the latter would become exponentially sized, in the number of variables. The effect of variable orders in BDDs is well known [56].

In the figure, the value $f(x)$ of an assignment x can be found by traversing the diagram as follows. One starts at the root node. A node is labeled with a variable x_i; if $x_i = 0$, we traverse the low (dotted) edge; otherwise, if $x_i = 1$, we traverse the high (solid) edge, until we arrive at a Leaf. To avoid cluttering the diagram, edges to the FALSE Leaf are not drawn in the figure.

A reduced and ordered BDD (ROBDD) is a canonical representation of its corresponding Boolean function [67]. From now on, we assume all BDDs are ROBDDs. Bryant [67] observed that such BDDs can be queried and manipulated in polynomial time in the size of the diagrams (number of nodes). For example, given BDDs f and g, with k and m nodes, respectively, a BDD representing the function $f \wedge g$ can be constructed and the number of models of f (i.e., $\vec{x}$ s.t. $f(x) = 1$) can be computed in $\mathcal{O}(km)$ and $\mathcal{O}(k)$ time, respectively. *Layer* i in an ordered BDD, is the set of nodes with variable label x_i (possibly empty).

A DSDBDD [45, 46, 92, 218, 264, 265, 282, 283] augments a BDD by considering the disjoint support decompositions of its nodes. A *disjoint support decomposition* of a function f decomposes it into its *kernel* k and its *factors* $j_1, \ldots, j_m$, as follows:

$$f(x_1, \ldots, x_n) = k(\ell_1, \ldots, \ell_m) \text{ with } \ell_i \triangleq j_i(x_{i,1}, \ldots, x_{i,n_i}) \tag{6.1}$$

Here factor j_i takes n_i variables as input; the variables ℓ_i are "dummy variables". The factors have no variables in common, so the numbers n_i sum to n. A decomposition is non-trivial if there are at least two factors, and one factor reads at least two variables.

Ashenhurst [22] was the first to develop a theory of disjoint support decompositions of Boolean functions and to give an algorithm which finds the decomposition given f's truth table, requiring time exponential in the number of variables. He showed that by repeatedly decomposing the functions k and j_i, the fixpoint reached is uniquely determined by f, up to complementation of the factors, and up to permutation of the order in which they appear as inputs to the kernel. This tree of functions is sometimes called the *Ashenhurst-Curtis decomposition* of f.

In [45], Bertacco and Damiani describe and implement an efficient algorithm to build a DSDBDD as follows. If a Shannon node in a BDD represents a function which allows a non-trivial decomposition, this node and its children are replaced by a dedicated *decomposition node* pointing to BDDs representing its kernel and its factors. These factors may themselves be decompositions, allowing 'nesting' of decompositions. This process is repeated until no Shannon node is eligible. Thus, a hybrid diagram is obtained, in which some nodes indicate decompositions (see Definition 6.2). Because of Ashenhurst's unique decomposition theorem [22], DSDBDDs are canonical like BDDs. The goal is that the new diagram is smaller than BDD, since this method may remove more nodes than it adds, but analytically little was known about this up to now.

Definition 6.2 (Disjoint Support Decomposition Diagram (DSDBDD))**.** A DSDBDD is a BDD whose internal nodes are either Shannon nodes or *decomposition nodes*. A decomposition node v has an outgoing edge to an internal node v_{ker} called its *kernel* and outgoing edges to its *factors* $v_1, \ldots, v_m$. It represents the function $[\![v]\!] = [\![v_{\text{ker}}]\!]([\![v_1]\!], \ldots, [\![v_m]\!])$, like in Equation 6.1. The diagram satisfies the following three rules:

1. If v is a factor of a decomposition node, then v satisfies $[\![v]\!](0, \ldots, 0) = 1$
2. Two factors $[\![v_i]\!]$ and $[\![v_j]\!]$ of a decomposition node have disjoint support, i.e., $vars([\![v_i]\!]) \cap vars([\![v_j]\!]) = \emptyset$, for $i \neq j$, where $vars(f)$ denotes the set of variables on which f depends.
3. The factors $v_1, \ldots, v_m$ of a decomposition node satisfy $\min vars([\![v_i]\!]) < \min vars([\![v_j]\!])$ for $i < j$, where min is relative to the diagram's variable order. ◇

Figure 6.1(c) shows a DSDBDD for the same function f as 6.1(a) and 6.1(b). Since this function can be expressed as a formula referencing each variable once, the DSDBDD can easily decompose it, obtaining the kernel k = AND on three variables. The factors are $x_i \Leftrightarrow x_{i+1}$ for $i = 1, 3, 5$. We remark that this succinct decomposition is available to a DSDBDD regardless of the variable order, whereas the BDD may have exponential size unless the right variable order is found. Figure 6.1(d) shows that the root of a DSDBDD is not necessarily a decomposition node.

Let us briefly motivate the three rules in Definition 6.2, which are similar to those formulated by Bertacco and Damiani [45]. The purpose of the rules is to keep the query and manipulation operations tractable, i.e., to prevent the diagram from becoming more expressive than intended. Notably, without rule 2, we no longer have efficient algorithms for querying and manipulating such a diagram; for example, model counting would be NP-hard, because, a 3-CNF formula may now be represented as a decomposition with kernel AND, and whose factors are disjunctions on three variables. Rule 1 compensates the fact that, according to Ashenhurst's Theorem, a decomposition is unique up to complementation of the factors. For example, if a function f has a decomposition $f = k(\ell_1, \dots, \ell_m)$ with $\ell_i = j_i(x_{i,1}, \dots, x_{i,n_i})$ as in Equation 6.1, then another decomposition is $f = k'(\neg\ell_1, \dots, \neg\ell_m)$, where k' takes the values $k'(\ell_1, \dots, \ell_m) \triangleq k(\neg\ell_1, \dots, \neg\ell_m)$. More generally, for each factor, the complementation may be chosen independently, leading to exponentially many possible decompositions. Rule 1 uniquely determines the choice of complementation by enforcing that, for each factor, $j_i(0, \dots, 0) = 1$. Similarly, rule 3 compensates for the fact that a decomposition is unique up to permutation of the kernel's input variables. For example, we may write the function f above as $f = k''(j_m, \dots, j_1)$ where $k''(\ell_1, \dots, \ell_m) \triangleq k(\ell_m, \dots, \ell_1)$.

Technically, the kernel of a decomposition node takes as input variables that are not inputs to f. The question which variables of the kernel to identify with which variables of the DD can be an important design decision for DSDBDD package implementations, and for obtaining canonicity. The diagram can be made canonical by imposing additional rules. Since such a canonical diagram is included in the above definition, a separation between BDDs and Definition 6.2 implies a separation with the canonical version. Therefore, we omit the strengthening of Definition 6.2 to obtain canonicity for the purposes of this work.

DSDBDDs supports the same queries and manipulation operations as BDDs (i.e., conjunction, disjunction, negation, model counting, etc.). These algorithms greedily minimize the DD by checking, whenever a new node is constructed, whether the node allows a decomposition, and then building this decomposition before proceeding. The worst-case running times of the algorithms are polynomial in the size of the BDDs (but not necessarily in the size of the DSDBDDs). In the best case, the running time is much better; in that case, the operands of, e.g., CONJOIN, are two decompositions whose kernels read exactly the same factors. In this case the operation can take advantage of the fact that, if $j_1, \ldots, j_m$ are functions such that f_1, f_2 decompose as $f = k_1(j_1, \ldots, j_m)$ and $f_2 = k_2(j_1, \ldots, j_m)$, then

$$f_1 \wedge f_2 = (k_1 \wedge k_2)(j_1, \ldots, j_m) \tag{6.2}$$

This allows the CONJOIN algorithm to work only on k_1 and k_2, whose diagrams may be exponentially smaller than the BDDs of f and g. In the worst case, however, the decompositions share no factors, so that CONJOIN must "unfold" these decomposition nodes into BDDs and the operation is done on the BDDs; hence, the running time is polynomial in the size of the BDDs. Bertacco and Plaza implemented these operations in the publicly available software package STACCATO [264,265]. They find that their package is competitive with CUDD both in terms of time and memory, on the task of compiling a Boolean circuit into a DD.

Similar ideas appear in AND/OR multi-valued DDs (AOMDDs) [215], which are canonial, and in BDS-Maj diagrams [11]. In BDS-Maj, the kernel is always chosen to be the Majority function on three inputs, and the factors may share variables, unlike in a DSDBDD.

6.3 Succinctness separation between DSDBDD and BDD

Theorem 6.1 shows an example of a *separating function* g (Equation 6.4) which has a small DSDBDD but exponential-sized BDD, *for every variable order*. It is based on three multiplexed copies of the order-parameterized function f, with variable orders π_0, π_1, π_2. By abuse of notation, we use z both as a bit-string, and as the integer $z \in \{0, 1, 2\}$ which the bit-string represents in base 2. The function f is well known to yield exponential BDDs for non-interleaved variable orders, as our generalized Lemma

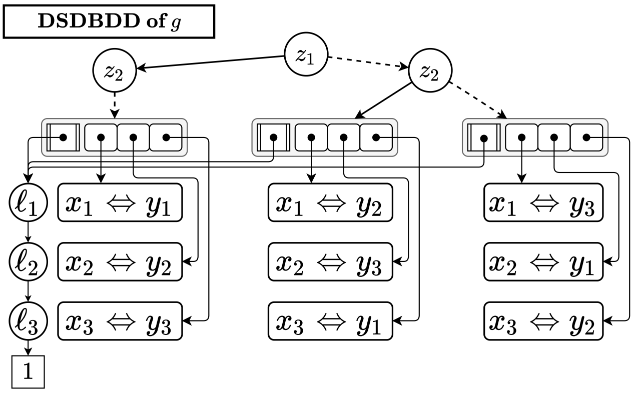

DSDBDD of g
z_1
z_2
z_2
ℓ_1
ℓ_2
ℓ_3
1
$x_1 \Leftrightarrow y_1$
$x_2 \Leftrightarrow y_2$
$x_3 \Leftrightarrow y_3$
$x_1 \Leftrightarrow y_2$
$x_2 \Leftrightarrow y_3$
$x_3 \Leftrightarrow y_1$
$x_1 \Leftrightarrow y_3$
$x_2 \Leftrightarrow y_1$
$x_3 \Leftrightarrow y_2$

Figure 6.2: The DSDBDD of the function g, in Equation 6.4 when $n = 3$. The permutations used are $\pi_0 = (1)(2)(3), \pi_1 = (1,2,3), \pi_2 = (1,3,2)$. The rectangle containing $x_i \Leftrightarrow x_j$ represents the BDD of the function $x_i \Leftrightarrow x_j$, as shown in Figure 2.3(e).

6.1 shows. We state it without proof.

$$f[\pi](x_1,\dots,x_n,y_1,\dots,y_n) \triangleq (x_1 \Leftrightarrow y_{\pi(1)}) \wedge \cdots \wedge (x_n \Leftrightarrow y_{\pi(n)}) \tag{6.3}$$

$$g(z,x_1,\dots,x_n,y_1,\dots,y_n) \triangleq f[\pi_z](x_1,\dots,x_n,y_1,\dots,y_n) \text{ for } z \in \{0,1,2\}. \tag{6.4}$$

Lemma 6.1. Let $\pi \in S_n$ and σ be an order over $\{x_1,\dots,x_n,y_1,\dots,y_n\}$ (the variables $f[\pi]$). For $1 \leq i \leq n$, say that x_i and $y_{\pi(i)}$ are *partners*. Let L be the first n variables according to σ. If k elements in L have their partner outside of L, then a BDD of $f[\pi]$ with variable order σ has at least 2^k nodes on layer n.

By choosing distinct permutations π, π', the functions $f[\pi], f[\pi']$ will disagree on which variables are partners. Theorem 6.1 shows that there exist many irreconcilable choices for permutations $\pi_0 - \pi_2$ in g, because the corresponding "partner graph", connecting two partner variables according to either permutation, is an *expander*, i.e., has high connectivity.

Theorem 6.1. Let π_0, π_1, π_2 be permutations chosen uniformly and independently at random from S_n. Then it holds that, with high probability, for every variable order σ over $\{x_1,\dots,x_n,y_1,\dots,y_n\}$, at least one of the BDDs for $f[\pi_0], f[\pi_1], f[\pi_2]$ has size $2^{\Omega(n)}$ and hence the BDD for g is also large.

Proof. Let G be the undirected bipartite graph with nodes $V = \{x_1,\dots,x_n,y_1,\dots,y_n\}$

and edges $E = E_0 \cup E_1 \cup E_2$ with $E_j = \{(x_i, y_{\pi_j(i)}) \mid 1 \leq i \leq n\}$. Then G is an expander with high probability by Theorem 4.16 in [164]. That is, there is a constant $\varepsilon > 0$ (independent of n) such that, with high probability, for all sets of vertices $L \subset V$, if $|L| \leq n$, then

$$\frac{|N(L) \setminus L|}{|L|} \geq \varepsilon \qquad \text{where } N(L) = \{w \mid \exists v \in L : (v, w) \in E\} \tag{6.5}$$

Let σ be a variable order of V (the variables of the functions $f[\pi_j]$), and let L be the first n variables according σ. Then there are at least $\varepsilon \cdot n$ vertices in $\overline{L}$ connected to L. Since each vertex is connected to at most 3 edges, it holds that one of the edge sets E_j is responsible for at least $\varepsilon \cdot n/3$ edges crossing over from L to $\overline{L}$. Let $K = E_j \cap (L \times \overline{L})$ be a set of pairs $(x_i, y_{\pi_j(i)})$ such that x_i is in L, but its partner $y_{\pi_j(i)}$ is $\overline{L}$. It follows from Lemma 6.1 that the corresponding function $f[\pi_j]$ has a BDD of size at least $2^{|K|} = 2^{\Omega(n)}$. Since $g_{|z:=j} = f[\pi_j]$, and since a BDD is at least as large as the BDDs of its subfunctions, g also has at least $2^{\Omega(n)}$ nodes. This holds w.h.p. over the choice of permutations. ☐

The DSDBDD of g is shown in Figure 6.2, for $n = 3$. For larger n, the DSDBDD simply has more “rows”, i.e., there are still three decomposition nodes, and they have n factors. The DSDBDD of g therefore has only $\mathcal{O}(n)$ nodes for larger n.

An immediate corollary is that the same relation holds between DSDBDDs versus ZDDs [229], Tagged BDDs [328] and CBDDs [68], since these decision diagrams are all at most a factor n smaller than BDDs on any function.

6.4 Conclusion and future work

We have analyzed the Disjoint Support Decomposition Binary Decision Diagram and found that it strictly dominates BDD and ZDD in terms of memory, up to polynomial overhead. That is, DSDBDDs can be exponentially smaller than BDDs. It remains an open question how DSDBDDs relates to other very expressive DDs; notably, it would be good to know its relation to SDDs, FBDDs, non-deterministic BDDs ($\vee$-BDD [54,55]) and d-DNNF. In addition, it would be interesting to map the complexity of DSDBDDs of the different operations considered by Darwiche & Marquis [97].

To the best of our knowledge, DSDBDDs have not been deployed on large, real-world problems as encountered, e.g., in model checking and synthesis. Given that we showed that DSDBDDs can be exponentially more succinct, and they retain canonicity of BDDs, it could be worthwhile to test the scalability of DSDBDD in practice. In a similar vein, the integration of disjoint support decompositions into other decision diagrams could be considered. Minato [230] shows how to find the DSDs of the nodes in a ZDD; a next step would be to integrate this into the Boolean operations of ZDDs, as was done in [264,265], so that the diagram remains small during compilation. Other promising candidates for integration with DSDs are FDDs and SDDs; we are not aware of any work in this direction.

Chapter 7

Model Checking with Sentential Decision Diagrams

We demonstrate the viability of symbolic model checking using Sentential Decision Diagrams (SDD), in lieu of the more common Binary Decision Diagram (BDD). The SDD data structure can be up to exponentially more succinct than BDDs, using a generalized notion of variable order called a variable tree ("vtree"). We also contribute to the practice of SDDs, giving a novel heuristic for constructing a vtree that minimizes SDD size in the context of model checking, and identifying which SDD operations form a performance bottleneck.

Experiments on 707 benchmarks, written in various specification languages, show that SDDs often use an order of magnitude less memory than BDDs, at the expense of a smaller slowdown in runtime performance.

This chapter contributes to Research Question 3:

> **Research question 3.** *Which classical decision diagrams might be effective for the analysis of quantum algorithms, if they were suitably adapted?*

In this case: can we draw upon the ideas behind Sentential Decision Diagrams to develop more powerful quantum DDs? To some extent, this has already happened: both (i) Tree Tensor Networks (TTN) and (ii) Weighted Context-Free Language Ordered Binary Decision Diagrams (WCFLOBDD) are tree-like structures which can be considered quantum extensions of SDDs. Interestingly, although the TTN predates SDDs, a straightforward “quantum SDD” has not yet been formulated or implemented. If this chapter offers any lessons to learn for TNNs and WCFLOBDDs, then perhaps it is that dynamic variable reordering (or, in this case, dynamic *tree reordering*) can help to both significantly speed up the algorithms and compress the data structure.

7.1 Introduction

Model checking is an automated way for verifying whether a system satisfies its specification [28]. Binary Decision Diagrams [67] (Binary DDs or BDDs) first revolutionized symbolic model checking [221]. But the representation may explode in size on certain instances [51, 97]. Later SAT and SMT based techniques, such as BMC [51] and IC3 [60], have been shown to scale even better.

DDs have the unique capability to manipulate (Boolean) functions. Due to this property, they can be used to encode a system in a bottom-up fashion [322]. In model checking, for example, this can be exploited to ‘learn’ transition relations on-the-fly [178], to soundly [207] implement the semantics of the underlying system. This fundamental difference from SAT also makes DDs valuable in various other applications [44, 243, 275].

To improve the performance of DDs, various more concise versions have been proposed [26, 68, 111, 229, 235, 245, 328]. One such data structure is the Sentential Decision Diagram (SDD) [96], recently proposed by Darwiche. SDDs subsume BDDs in the sense that every BDD is an SDD, and SDDs support the same functionality for function manipulation. Specifically, SDDs support the operations of conjunction, disjunction, negation operations, and existential quantification, making them suitable for the role of data structure in symbolic model checking. Since every BDD is an SDD, they need not use more memory than BDDs, and if an SDD’s *variable tree* (vtree) is chosen well, then an SDD has the potential to use exponentially less memory than a BDD [58]. While SDDs have already shown potential for CAD applications [83], they have not yet been tried for model checking. And because of the relevance of DDs in many other applications, an evaluation of the performance of SDDs is also of interest in its own right.

The biggest obstacle to overcome, therefore, is to find a good vtree for the SDD. A vtree is to an SDD as a variable order is to a BDD. In BDD-based symbolic model checking, the BDD's variable order determines the size of the BDD in memory, and the choice of variable order may determine whether or not the verifier runs out of memory.

We propose in Section 7.3 three heuristics for constructing vtrees. First, we propose that the vtree is constructed in two phases. In phase 1, we build an abstract variable tree which minimizes the distance between dependent variables in the system. In phase 2, the leaves that represent integer variables are "folded out", and are replaced by "augmented right-linear" vtrees on 32 variables. These heuristics *statically* construct a vtree, before model checking, so that the vtree is preserved throughout the analysis.

We implemented our approach in the language-independent model checker model checker LTSmin [178][*], using the SDD package provided by the Automated Reasoning Group at UCLA [25]. To evaluate the method, we test the performance of SDDs on 707 benchmarks from a diverse set of inputs languages: DVE [31] (293), Petri nets [187] (357), Promela [159] (57). We test several configurations, corresponding to using all heuristics above, or only a subset, with and without dynamic vtree search (i.e., modifying the vtree during model checking). These languages represent very different domains. By obtaining good results across all domains, we gain confidence that the advantage of SDDs over BDDs is robust.

Experiments show that SDDs often use an order of magnitude less memory than BDDs on the same problem, primarily on the more difficult instances. Among instances that were solved by both SDDs and BDDs, 75% were solved with a smaller SDD, and among those instances, the SDD was on average 7.9 times smaller than the BDD. On the other hand, the SDD approach is up to an order of magnitude slower than the BDD; 82% of the instances were solved faster with BDDs, with an average speedup of 2.7. We investigate the causes of this slowdown in SDDs, but also demonstrate with larger benchmarks, that memory/time trade-off is worthwhile as we can solve larger instances with SDDs. Our analysis shows that the set intersection and union operations in SDDs are relatively fast, whereas existential quantification can be a performance bottleneck. This is despite the recent discovery that set intersection and union have exponential worst-case behaviour in theory [322].

---

[*] Our implementation's source code is available at https://doi.org/10.5281/zenodo.3940936

7.2 Background

7.2.1 Sentential Decision Diagrams

A Sentential Decision Diagram (SDD), recently proposed by Darwiche [96], is a data structure which stores a set $K \subseteq \{0,1\}^n$ of same-length bit vectors. In our context, a bit vector represents a state of a system, and variables are represented using multiple bits. Like Binary Decision Diagrams (BDDs) [68], SDDs are a canonical representation for each set and allow, e.g., union and intersection, with other sets. Following is a succinct exposition of the data structure.

Suppose we wish to store the set $K \subseteq \{0,1\}^n$ of length n bit vectors. If $n = 1$, then K is one of four base cases, $K \in \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$. Otherwise, if $n \geq 2$, then we decompose K into a union of simpler sets, as follows. Choose an integer $1 \leq a < n$. Split each string into two parts: the first a bits and the last $n-a$ bits, called the prefix and the postfix.

For a string $p \in \{0,1\}^a$, let $Post(p) \subseteq \{0,1\}^{n-a}$ be the set of postfixes $s \in \{0,1\}^{n-a}$ with which p can end. More precisely:

$$Post(p) = \{s \in \{0,1\}^{n-a} \mid ps \in K\} \tag{7.1}$$

where ps is the concatenation of p with s.

Next, define an equivalence relation on the set $\{0,1\}^a$ where $p_1 \sim p_2$ when $Post(p_1) = Post(p_2)$. Suppose that this equivalence relation partitions the set $\{0,1\}^a$ into the ℓ sets $P_1, \ldots, P_\ell$. Let $\overline{p_1} \in P_1, \ldots, \overline{p_\ell} \in P_\ell$ be arbitrary representatives from the respective sets. Then we can represent K as

$$K = \bigcup_{i=1}^{\ell} P_i \times Post(\overline{p_i}) \tag{7.2}$$

The sets P_i, P_j are disjoint for $i \neq j$, because these sets form a partition of $\{0,1\}^a$, so the Cartesian products $P_i \times Post(\overline{p_i}), P_j \times Post(\overline{p_j})$ are also disjoint for $i \neq j$. Hence we have decomposed K into a disjoint union of smaller sets.

To build an SDD for K, recursively decompose the sets $P_1, \ldots, P_\ell$ and $Post(\overline{p_1}), \ldots, Post(\overline{p_\ell})$, until the base case is reached, where the bit vectors have length 1. To this end, choose two integers $1 \leq a_1 < a$ and $a \leq a_2 < n$ and decompose the

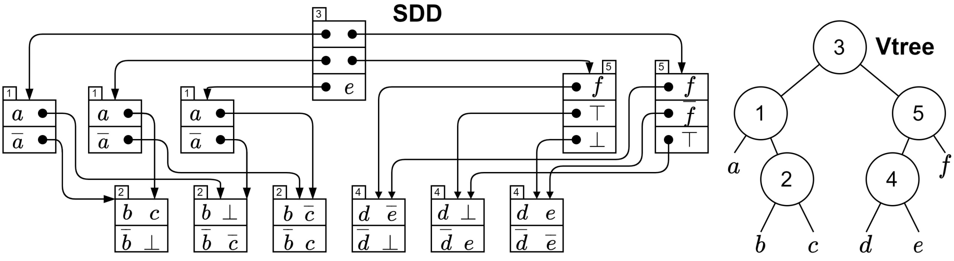


Figure 7.1: An SDD and the vtree it is normalised to. Rectangles are SDD nodes, labelled with the vtree node they are normalized to.

sets P_i with respect to a_1 and decompose the sets $Post(\overline{p_i})$ with respect to a_2. In the literature, the sets P_i are called the *primes*, the sets $Post(\overline{p_i})$ the *subs*, and the sets $P_i \times Post(\overline{p}_i)$ are called the *elements* of K.

Example 7.1. Figure 7.1 shows an SDD for a set of length-6 bitstrings, normalized to the vtree on the right. A rectangle represents a set. A box within a rectangle represents an element of that set, with its prime on the left and its sub on the right. The box represents the Cartesian product of its prime and sub. In particular, the SDD nodes' labels indicate which vtree node they are normalized to. A $\top$ represents $\{0, 1\}$, and $\bot$ represents the empty set.

Variable trees. In the construction of the SDD, the choice of the integer a determines how the top level of the SDD decomposes into tuples. The left and right partitions are then further decomposed, recursively, resulting in a full binary tree, called a *variable tree*, or "vtree" for short. This binary tree is full in the sense that each node is either a leaf, labelled with a single variable, or else it is an internal node with two children (internal nodes do not correspond to variables). An SDD's variable tree, then, determines how the smaller sets decompose and therefore completely determines the SDD. SDDs are called a *canonical representation* for this reason. Consequently, searching for a small SDD reduces to searching for a good vtree.

A left-to-right traversal of a vtree induces a variable order. When we say that an algorithm for vtree optimization preserves the variable order, we mean that the algorithm produces a new vtree with the same induced variable order. A *right-linear* vtree is one in which every internal node's left child is a leaf. BDDs are precisely those SDDs with a right-linear vtree, hence SDDs generalize BDDs.

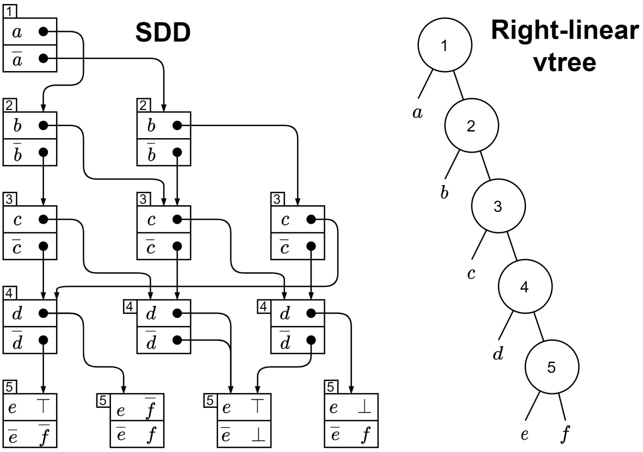


Figure 7.2: An SDD normalized to a right-linear vrtee.

Example 7.2. Figure 7.2 shows an SDD normalised to a right-linear vtree for the same set K as in Example 7.1. The label in a round node indicates the vtree node it is normalised to. Although the vtrees are different, they induce the same variable order $A < B < C < D$. This SDD has six nodes, whereas the SDD in Figure 7.1 has seven nodes. We see, therefore, that the variable tree can affect the size of the SDD, independently of the variable order.

Formal semantics An SDD node t formally represents a set of strings denoted by $\langle t \rangle$, as follows. The node may be of one of two types:

1. t is a terminal node labelled with a set $A \in \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$, in which case $\langle t \rangle = A$.
2. t is a decomposition node, and $t = \{(p_1, s_1), \ldots, (p_\ell, s_\ell)\}$, where p_i and s_i are other SDD nodes. Then $\langle t \rangle = \bigcup_{i=1}^{\ell} \langle p_i \rangle \times \langle s_i \rangle$

7.2.1.1 Queries and transformations

Because SDDs are a canonical representation, equivalence of two SDDs can be checked in constant time, provided that they obey the same variable tree. SDDs support polynomial-time model counting and model enumeration. Whether SDDs support polynomial-time conjunction and disjunction was an open question which was recently answered in the negative, unfortunately, by Van den Broeck and Darwiche [322].[†] However, they find empirically that this worst-case behaviour rarely manifests in practice.

7.2.2 Model checking and reachability

Model checkers verify whether the behavior of a system M conforms to a specification φ. To support reactive systems, the property φ is expressed in a temporal logic, such as, LTL [266], CTL [85] or the modal μ-calculus [189]. The formal check involves deciding semantic entailment, i.e., $M \models \varphi$, and can be implemented as a fixpoint computation on the semantic interpretation of the system [28], typically a transition system (Kripke structure). Since the fixpoint computations reduce to reachability, we only need to focus on solving reachability to evaluate and compare the performance of various decision diagrams in model checking.

The reachability procedure is a repeated image computation with the symbolic transition relation R until a *fixpoint* is reached. Starting from $A := S_0$, we compute $A_{i+1} := A_i \cup R(A_i)$, until this computation converges, i.e., an iteration adds no more states. In each iteration, the set A_i contains a set of system states.

In symbolic model checking, we store the set A_i using a decision diagram, e.g., an SDD. In order to encode the relation R, the vtree will contain, for every variable $x \in X$, a "primed copy" $x' \in X'$. The SDD encoding the transition relation will contain one satisfying assignment for each transition $(s, t) \in R$ (and no more). We therefore denote this SDD as $R(X, X')$; and similarly the SDDs representing sets as $A(X)$. For reachability with SDDs, we need to compute the image function of the transition relation. Equation 7.3 shows how the image is computed using elementary manipulation operations on SDDs (conjunction, existential quantification and variable renaming). The conjunction constrains the relation to the transitions starting in A; the existential quantification yields the target states only and the renaming relabels target states in X instead of X' variables.

$$\text{Image}(R, A) \triangleq (\exists X \colon R(X, X') \wedge A(X))[X' := X] \tag{7.3}$$

[†] More precisely, they show that exponential blowups may occur when disjoining two SDDs if the output SDD must obey the same vtree, but it remains an open question whether the blowup is unavoidable when the vtree is allowed to change.

In our context, we have the specification of the system we wish to model check. Consequently, we know in advance which variables and actions exist, (e.g., in a Petri net the variables are the places and the actions are the transitions), and therefore we know which read/write dependencies exist between actions and variables.[‡] We use this information to design heuristics for SDDs. The integers in our systems are at most 16-bit, so each integer is represented by 16 primed and unprimed variables in the relation (for details, see Section 7.5).

7.3 Vtree heuristics

We propose three heuristics for constructing vtrees based on the read/write dependencies in the system being verified. Sec. 7.3.1 proposes that the vtree is constructed in two phases, and Sections 7.3.2 and 7.3.3 propose heuristics to be used in the two respective phases. Since the vtree structure can influence SDD size independent of the variable order, as shown in the previous section, we assume a good variable order and give heuristics which construct a vtree which preserves that variable order. The input to our vtree construction is therefore a good variable order, provided by existing heuristics [224], along with the known read/write dependencies between those variables, derived from the input that we wish to model check. We then construct the vtree using the heuristics explained below, preserving the variable order. The resulting vtree is then preserved throughout the reachability procedure, or, if dynamic vtree search is enabled, serves as the initial vtree.

Our application requires that we store relations rather than states, so our vtrees will contain two copies of each variable, as explained in Sec. 7.2.2.

7.3.1 Two-phase vtree construction

Any vtree heuristic will have to deal with the fact that the system's variables are integers, whereas an SDD (or a BDD) encodes each bit as an individual variable, so that an integer is represented by multiple variables. Therefore, our first heuristic is that we propose that a vtree be constructed in two phases.

[‡] While for some specification languages this information has to be estimated using static analysis, there are ways to support dynamic read/write dependencies [223], but for the sake of simplicity, we do not consider them here.

In the first phase, one builds a small vtree called an “abstract variable tree”, containing all of the system’s variables as leaves. In the second phase, we “fold out” the leaves corresponding to variables that represent more than one bit, i.e., whose domain is larger than $\{0, 1\}$. Folding out a leaf means replacing it by a larger vtree on several variables. For example, a leaf representing a 32-bit integer variable will be folded out into a vtree on 64 variables (namely 32 state variables and 32 primed copies). Figure 7.3 illustrates this process. In this example, a system with four 4-bit integer variables, p, x, y, z, may lead, during phase 1, to the abstract variable tree in the top of the picture. In phase 2, the leaves may be folded out into, in this case, vtrees on 8 bits, producing either the bottom left or right vtree.

Note that this never produces a right-linear vtree, even if both the abstract variable tree and the integer’s vtree are right-linear.

7.3.2 Abstract variable tree heuristic

To design the abstract variable tree, we adopt the intuition that, if a system reads variable x in order to determine the value to write to variable y, then, all other things equal, it would be better if the distance in the vtree between x and y was small.

To capture this intuition, we define a penalty function $p(T)$ (Equation 7.4) on a vtree T, which we call the *minimum distance* penalty. We then try to minimize the value of this penalty function by applying local changes to the vtree. The penalty function will simply be the sum, over all read-write dependencies in the system, of the distance between the read and write variable in the vtree. By distance in the vtree, denoted d_T, we mean simply the length of the unique path in the abstract variable tree from the leaf labelled with variable x to the leaf labelled with variable y. Here A is the set of a system’s actions, R_a is the set of variables read by action a, and W_a is the set of variables written to by action a.

$$p(T) = \sum_{a \in A} \sum_{(r,w) \in R_a \times W_a} d_T(r, w) \tag{7.4}$$

This heuristic is inspired by various variable ordering heuristics, such as Cuthill-Mckee [90] and the bandwidth and wavefront reduction algorithms [224], all of which try to minimize the distances between dependent variables.

7.3.3 Folding out vtrees for large variable domains

We consider two options for folding out an abstract variable. First, such a variable may be folded out into a right-linear vtree, i.e., to a BDD. Figure 7.3 (left) shows this approach. Second, we introduce the "augmented right-linear vtree", which is a right-linear vtree except that each pair of corresponding state and prime bit is put under a shared least common ancestor. The idea is that a state and primed bit are more closely correlated to one another than to other bits. Figure 7.3 (right) shows this approach.

Note that in both options, the resulting vtree is not a right-linear vtree.

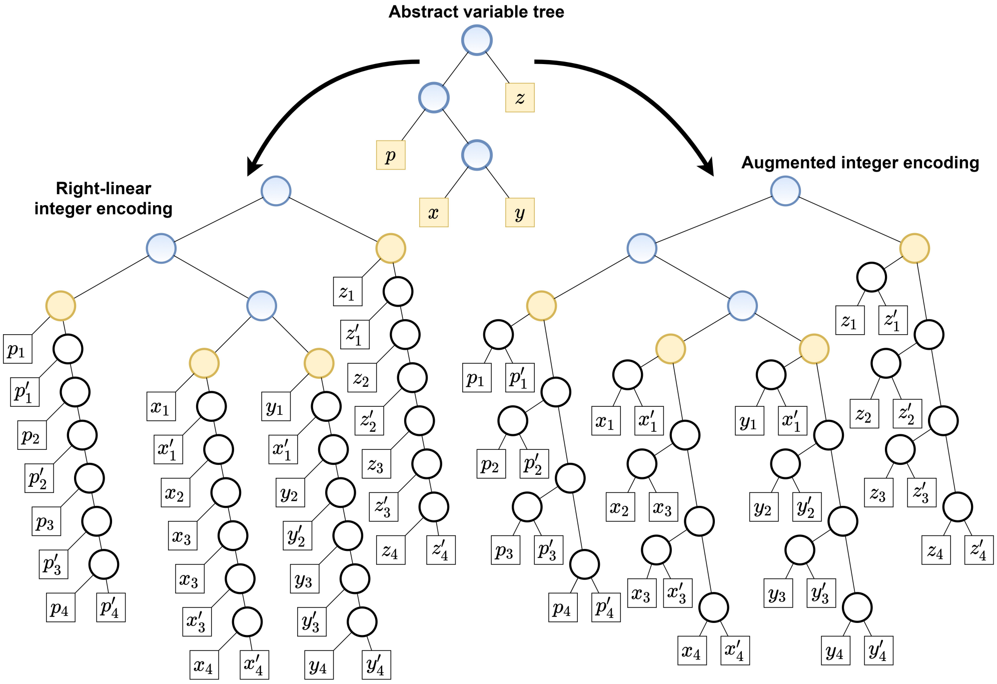

Abstract variable tree
Right-linear integer encoding
Augmented integer encoding
p
x
y
z
p_1 p'_1 p_2 p'_2 p_3 p'_3 p_4 p'_4
x_1 x'_1 x_2 x'_2 x_3 x'_3 x_4 x'_4
y_1 y'_1 y_2 y'_2 y_3 y'_3 y_4 y'_4
z_1 z'_1 z_2 z'_2 z_3 z'_3 z_4 z'_4

Figure 7.3: An abstract variable tree (top) with four 4-bit integers can be folded out in two ways: To a vtree with right-linear integer encoding (left) or with augmented integer encoding (right).

7.4 Related work

Many algorithms exist for static variable ordering for BDDs, e.g., the Cuthill-McKee algorithm [90], Sloan [297] and MINCE [9]; see [276] for a survey. These algorithms exploit event-locality: the observation that most events involve only very few of a system's variables. By using an order in which variables appear next to each other when they appear together in many events, the size of a BDD can decrease relative to variable orders which do not take this into account. Finding an optimal order is NP-Hard [56], so these algorithms heuristically minimize certain distance metrics [292].

We are not aware of any work which attempts to statically construct vtrees. Works in this direction are Darwiche [96] and Oztok and Darwiche [249], which show that a CNF formula with bounded treewidth, or CV-width, respectively, can be compiled into a polynomial-size SDD if a vtree is known which reflects the tree decomposition of the formula. Unfortunately, finding the tree decomposition is NP-Hard [19], and indeed even hard to approximate [353], although this can be done in linear time when the treewidth is bounded [53].

Choi and Darwiche [83] introduce *dynamic vtree search*, that is, they modify the vtree in between SDD operations, aiming to reduce the SDD size. This is useful even when space is not a bottleneck, because, all other things equal, algorithms run faster on small SDDs than on large ones. When dynamic vtree search is enabled in our experiments, we use the implementation of Choi and Darwiche.

7.5 Experiments

We evaluate our methods on 707 benchmarks, consisting of 293 DVE models, 357 Petri Net models and 57 Promela models. The benchmarks are provided[§] by Meijer and van den Pol [224], who collected them from the BEEM database [258] and the 2015 Petri Net Model Checking Competition [188], selecting those problems that can be handled by the static variable ordering algorithms. To utilize these different specification languages, we extended[¶] the LTSmin model checker [178] with an SDD package [25]. The language-independent interface of LTSmin [178] represents all variables as 16-bit integers. Both the BDD and the SDD packages benchmarked here utilize the same

---

[§] The benchmarks are available at `https://github.com/utwente-fmt/BW-NFM2016`

[¶] Again, the implementation is available at https://doi.org/10.5281/zenodo.3940936 interface. As explained in the introduction, we are interested in evaluating the performance of SDDs with respect to BDDs, to study the feasibility of SDDs for model checking and identify bottlenecks in SDD implementations.

To implement the abstract variable tree heuristic described in Sec. 7.3.2, we perform a vtree search before executing the reachability procedure in the model checker. We greedily perform local modifications on the vtree until either (i) no possible modification improves the value of the penalty function p (Equation 7.4), or (ii) our budget of n local modifications is exhausted, where n is the number of program variables, or (iii) 30 seconds have elapsed. The local modifications are restricted to *vtree rotations*, which is an operation which inverts the roles of parent and child for a particular pair of adjacent internal vtree nodes. This operation preserves the induced variable order.

We experiment with two BDD configurations and four SDD configurations, listed in Table 7.1. Each configuration is tested on all 707 benchmarks, with a timeout of 10 minutes (600 seconds) per benchmark. Experiments are done on a Linux machine with two 2.10 GHz Intel Xeon CPU E5-2620 v4 CPUs, which each have 20 MB cache and 16 cores, with 90GB of main memory. The BDDs are implemented using the Sylvan multicore BDD package (version 1.4.0) [327]. The SDDs are implemented using the SDD package (version 2.0) [25]. Before vtree construction, we apply the Cuthill-Mckee (BCM) [90] variable reordering heuristic provided by LTSmin [224]. The input to our vtree construction is the variable order found by BCM, along with the known read/write dependencies between those variables. We perform a small case study of six problems from the DVE set with a timeout of 3 hours instead of 10 minutes.

Table 7.1 lists the parameter settings used for benchmarks. The first row denotes the BDD configuration against which we compare our method. The SDD configurations are such that, in each next row, we “turn on” one more vtree heuristic. The BCM in the variable order column refers to the Cuthill-Mckee ordering heuristic [90], which is performed once, before vtree construction and model checking. The column “Augmented int” records whether integer leaves in the abstract variable tree are folded out to a right-linear or an augmented right-linear representation. The “vtree heuristic” denotes whether the abstract variable tree was right-linear (“No”) or was constructed using the minimum distance penalty (“Min. distance”). The row BDD refers to the single-threaded implementation, whereas the row BDD(32) uses the multi-threaded (32-core) implementation [327], which we include only for reference as SDDs [25] are still single-threaded.

Dynamic vtree search [83] is the SDD analog of dynamic variable reordering, that is, it modifies the vtree during execution, aiming to reduce the size of the SDD. The column "Dynamic vtree search" records whether vtree search was enabled during reachability analysis. In that case, at most half the time is spent on vtree search.

Configuration name	Variable order	Two-phase construction	Augmented int	vtree heuristic	Dynamic vtree search
BDD	BCM	(N/A)	(N/A)	(N/A)	No
BDD(32)	BCM	(N/A)	(N/A)	(N/A)	No
SDD(r)	BCM	Yes	No	No	No
SDD(r,a)	BCM	Yes	Yes	No	No
SDD(d,a)	BCM	Yes	Yes	Min. distance	No
SDD(d,a)+s	BCM	Yes	Yes	Min. distance	Yes

Table 7.1: The parameter settings used in the experiments.

The memory usage is computed as follows. A BDD node takes up 24 bytes. An SDD node takes up 72 bytes, plus 8 bytes per element (an SDD node has at least two elements). That this measure is fair for SDDs was verified by Darwiche in personal communication. We take this approach instead of relying on the memory usage as reported by the operating system, which is not indicative of the size of the SDD / BDD due to the use of hash tables. For each benchmark, we will report the peak memory consumption, that is, the amount of memory in use when the BDD / SDD was at its largest during the reachability procedure.

7.5.1 Results

Figure 7.4 and 7.5 show the results of the experiments. In each plot, a blue "×" represents a DVE instance, a green "△" a Petri net instance, and a red "+" a Promela instance. Instances on the black diagonal line are solved using an equal amount of time or memory by both approaches, instances on a gray diagonal line are solved by one method with an order of magnitude advantage over the other. All graphs are formatted such that benchmarks that were solved better by the more advanced configuration appear as a point below the diagonal line; otherwise, if the less advanced configuration performed better, then it appears above the diagonal line. The horizontal and vertical lines of instances near the top left of the figures, represent instances where one or both methods exceeded the timeout limit.

Figure 7.4 (top) shows that the simplest SDD configuration, SDD(r), tends to outperform BDDs in terms of memory, especially on more difficult instances, but not in terms of time, often taking an order of magnitude longer. As a result of the 600 second timeout, BDDs manage to solve many instances that SDDs fail to solve. The middle row shows that using augmented integer vtrees is not perceptibly better than using right-linear vtrees. The bottom row shows that using our minimum distance heuristic improves both running time and diagram size. Finally, Figure 7.5 (top) shows that adding dynamic vtree search yields even smaller diagrams, and on difficult instances, tends to be slightly faster.

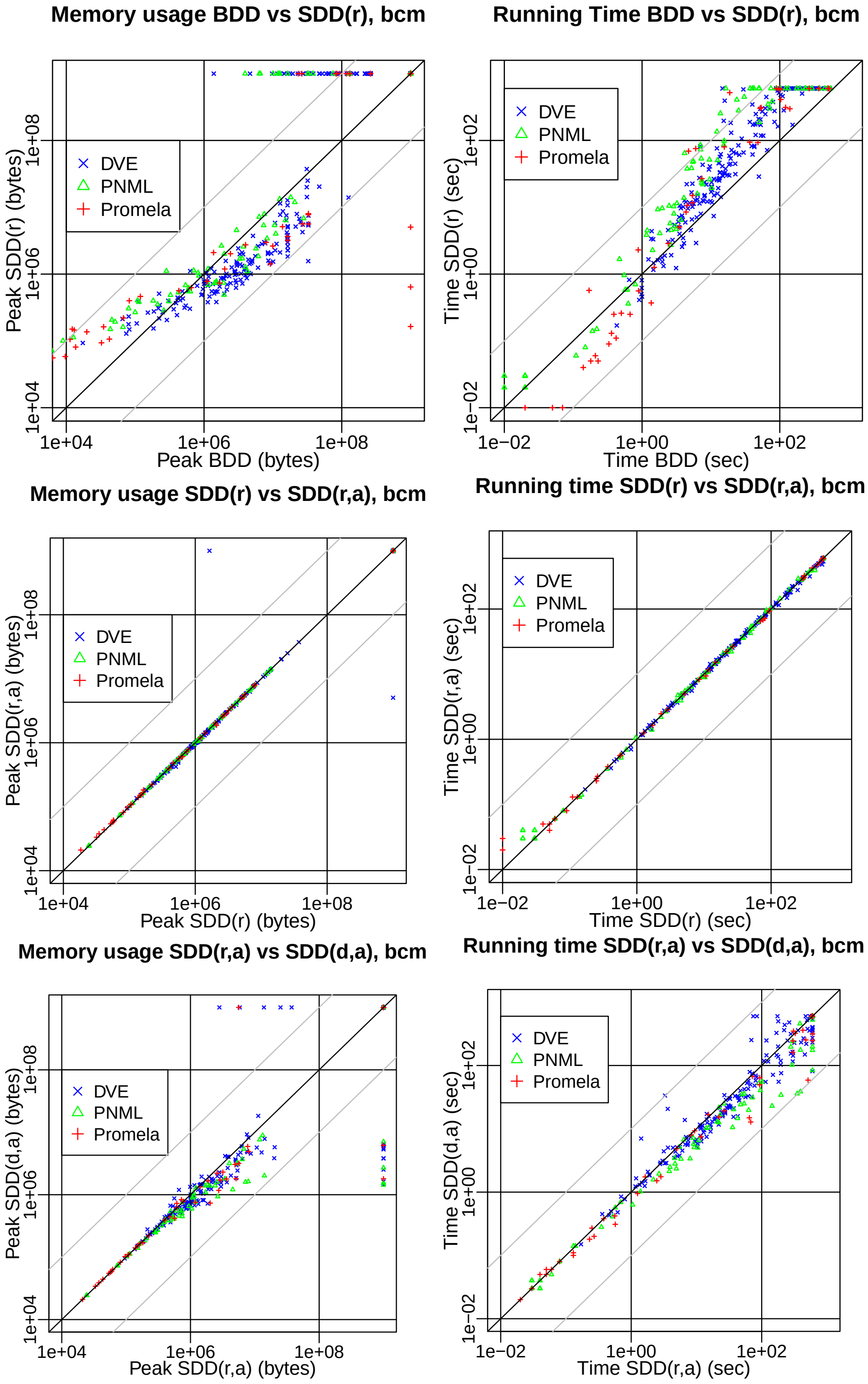
Figure 7.5 (bottom) compares SDD(d,a)+s to BDDs. It shows that memory usage is often an order of magnitude lower than that of BDDs on the same benchmark. The effect is even more pronounced than in Figure 7.4 (top). Unfortunately, this configuration often uses more time than BDDs do.

We see that two of our heuristics improve the performance of SDDs, at least as far as memory is concerned, namely, (i) using two-phase vtree construction instead of a BDD (Figure 7.4, top) and (ii) using the minimum abstract variable tree distance heuristic (Figure 7.4, bottom).

Table 7.2 and Table 7.3 show the relative speedup and memory improvements of all pairs of methods. A cell indicates the ratio by which the column method outperformed the row method, restricted to those benchmarks on which it outperformed the row method. For example, Table 7.2, shows that, on the 264 benchmarks that both SDD(r) and BDD solved, SDD(r) uses less memory than BDD on 172 benchmarks, and on those benchmarks, uses 3.1 times less memory on average, whereas on the remaining 92 instances, BDD used 5.5 times less memory than SDD(r).

From the tables we see that no method dominates another, even in just one metric. However, we do see that SDD(d,a)+s often uses much less memory than BDD (namely on 75% of benchmarks that both methods solved), and then on average by a factor 7.9. Strikingly, Figure 7.5 reveals that this advantage becomes progressively larger as the instances become more difficult. We also see that each heuristic helps a little bit, with the exception of augmented integer vtrees. That is, in the sequence BDD, SDD(r), SDD(d,a), SDD(d,a)+s, each next configuration, compared with the previous, is better on more instances than it is worse (There is one exception: SDD(d,a)+s is often smaller but slower than SDD(d,a)).

The results of the case study with a timeout of 3 hours are shown in Table 7.4, and are



Memory usage BDD vs SDD(r), bcm

Running Time BDD vs SDD(r), bcm

Memory usage SDD(r) vs SDD(r,a), bcm

Running time SDD(r) vs SDD(r,a), bcm

Memory usage SDD(r,a) vs SDD(d,a), bcm

Running time SDD(r,a) vs SDD(d,a), bcm

Figure 7.4: Memory (left) and time (right) consumption comparisons of various SDD configurations.

Table 7.2: Average ratio of memory usage on instances that the column method solved using less memory than the row method. Numbers in brackets indicate the number of benchmark instances on which the column method outperformed the row method. We leave out BDD(32), since its memory use is equal to BDD.

Memory	BDD	SDD(r)	SDD(r,a)	SDD(d,a)	SDD(d,a)+s
BDD	1	3.1 (172)	3.1 (173)	4.4 (193)	7.9 (220)
SDD(r)	5.5 (92)	1	1.02 (138)	1.5 (186)	2.3 (216)
SDD(r,a)	5.7 (91)	1.01 (128)	1	1.5 (182)	2.3 (218)
SDD(d,a)	6.2 (82)	1.1 (75)	1.1 (79)	1	1.6 (212)
SDD(d,a)+s	6.9 (72)	1.2 (48)	1.1 (46)	1.1 (62)	1

Table 7.3: Average ratio of time used on instances that the column method solved using less time than the row method. Numbers in brackets indicate the number of benchmark instances on which the column method outperformed the row method.

Time	BDD	BDD(32)	SDD(r)	SDD(r,a)	SDD(d,a)	SDD(d,a)+s
BDD	1	6.1 (321)	2.2 (45)	1.9 (46)	1.9 (54)	2.1 (51)
BDD(32)	1.2 (11)	1	2.4 (21)	2.1 (21)	2.1 (22)	2.1 (23)
SDD(r)	3.8 (219)	17.2 (243)	1	1.04 (121)	1.8(187)	2.1 (144)
SDD(r,a)	3.7 (218)	17.0 (243)	1.1 (145)	1	1.7 (189)	2.1 (149)
SDD(d,a)	2.8 (221)	13.2 (253)	1.6 (74)	1.5 (72)	1	2.0 (78)
SDD(d,a)+s	2.7 (240)	13.3 (269)	1.4 (120)	1.3 (115)	1.4 (196)	1

included in Figure 7.5 (bottom left). In this problem set, SDDs consumed 31 times less memory on average.

7.5.2 SDD runtime profiles

Figure 7.6 breaks up the time spent on the most difficult instances into the three SDD operations (Existential quantification, intersection and union of sets), and the glue code which connects LTSmin to the SDD package (LTSmin's Partitioned Next-State Interface (PINS), is described in [178]). For a few instances (`pouring`, `lup`, and `peg_solitaire`) the glue code takes the most runtime. We verified that in these cases the input system contains few locality (the PINS dependency matrix is dense), which is not what LTSmin was designed for, so we will not discuss these instances further.

In the SDD packages, we see that the main bottleneck is existential quantification and vtree search, whereas intersection and especially union take less time. The vtree search pays off, because in this setting the model checking procedure performed best as we discussed in the previous section. It comes as no surprise that existential quan-

Table 7.4: Time and peak memory consumption (kB) on six large problems. Times are in seconds.

Metric	Name	synapse.7	telephony.4	szymansky.5	sched_world.3	sokoban.1	telephony.7
Space	SDD(d,a)+s	6804	20893	24587	45694	1838	19453
	BDD	88905	131214	165836	259544	261113	262210
Time	SDD(d,a)+s	836	1078	4918	1789	858	2295
	BDD	73	19	42	41	28	36

tification is a bottleneck, because quantification for multiple variables is NP-Hard and all variables are bit-blasted in LTSmin. It is, in some sense, good news: There is a lot performance to be gained by eliminating this bottleneck, and there is good hope that this is possible, because NP-Hard problems can often be made tractable in many easy instances by developing good heuristics. The SDD package currently employs no special heuristics to perform existential quantification.

We were unable identify unique characteristics in terms of their number of variables or their number of actions for the outlier instances in which set intersection is the bottleneck (e.g., `elevator2.3`).

7.6 Conclusion and future work

Sentential Decision Diagrams are a viable alternative to Binary Decision Diagrams for use in symbolic model checking. The size in memory of an SDD is determined by its vtree, and choosing this vtree was the foremost challenge when implementing SDD-based model checking. This challenge was satisfactorily met with novel heuristics. Our experiments show that the novel heuristics yield SDDs that are often an order of magnitude smaller than BDDs on the same problem, and this advantage becomes larger on more difficult instances. These results are robust to the difficulty of the instance, as the performance of BDDs on our benchmark set spans five orders of magnitude, from 10kB of memory to 300MB of memory. That SDDs are slower contrasts with findings of Choi and Darwiche [83]. We suggest three avenues for research in the near future.

1. Combine variable order heuristics with vtree structure heuristics
2. Use more advanced data structures than SDD
3. Eliminate the speed bottlenecks in SDDs; specifically, develop (heuristics for) faster existential quantification algorithms.

For Item 2, the Zero-Suppressed Sentential Decision Diagram [245], and the Tagged Sentential Decision Diagram [111,328], look like promising next steps, because both do well when representing sparse sets, which is the case in the current application. Using Uncompressed Sentential Decision Diagrams looks promising in theory, but recent work suggests that much still needs to be done before they can be considered a tractable data structure [322].

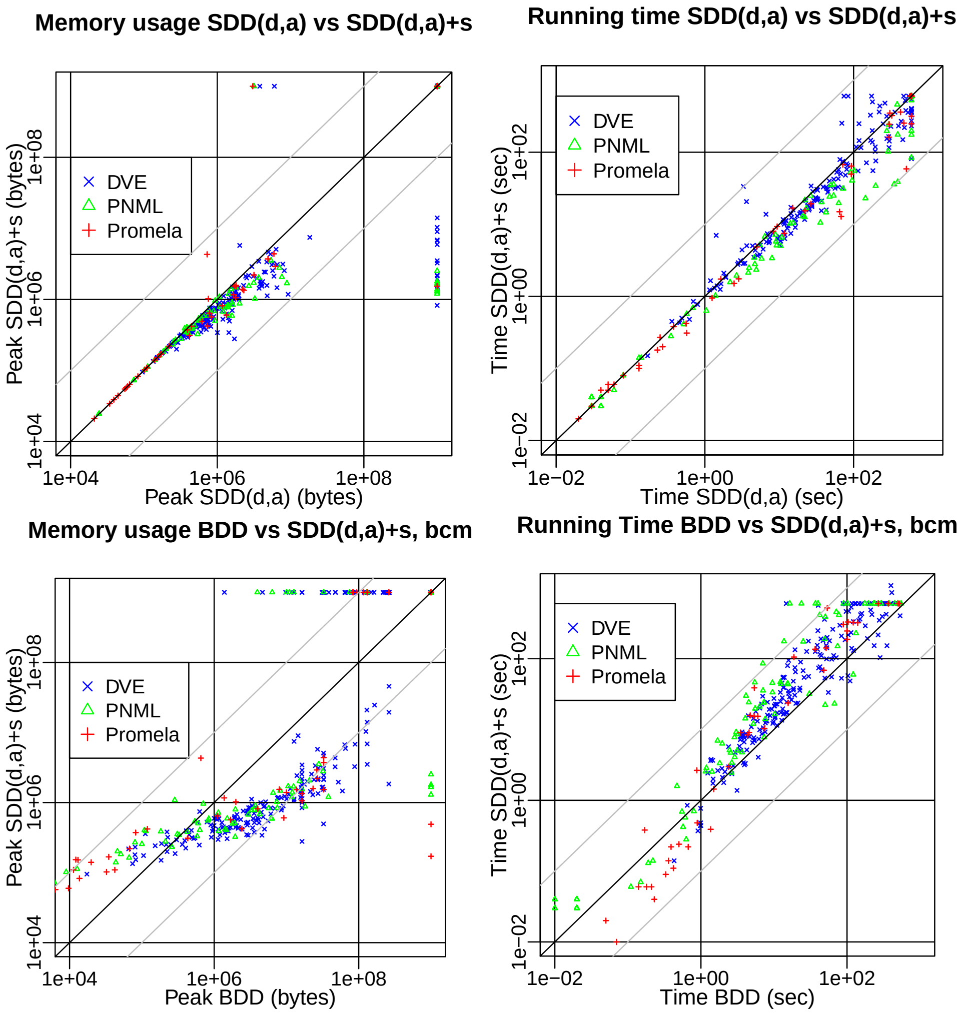


Figure 7.5: Memory (left) and time (right) consumption comparisons (continued).

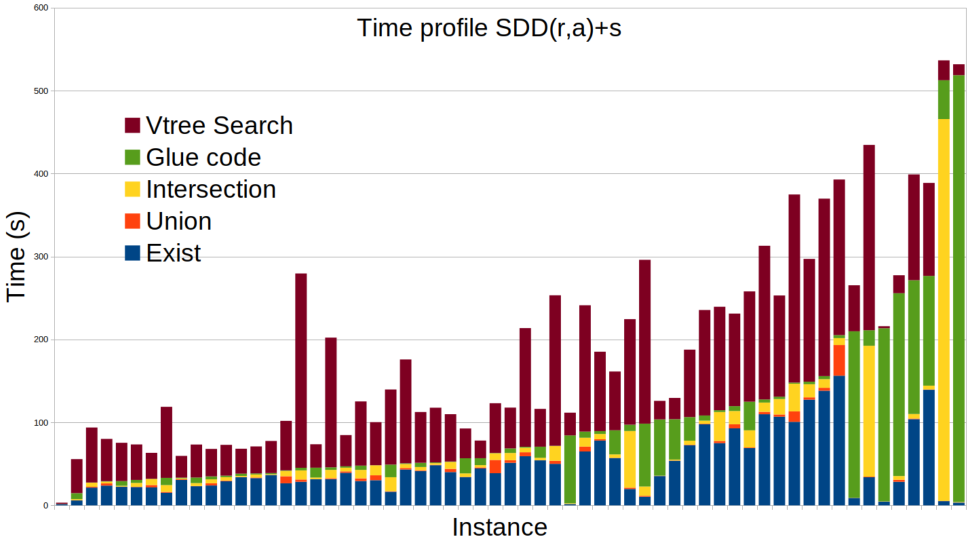


Figure 7.6: The time profile of SDD configuration SDD(d,a)+s, split among the three primary SDD set operations of Intersection, Union and Existential quantification, and the glue code which connects `LTSmin` to SDD. Shown are the DVE and PNML instances that were solved in more than 100 seconds. The instances are sorted by the time taken by the SDD operations, plus glue code.

Chapter 8

Conclusions and outlook

In this dissertation, we set out to develop methods for analysis and particularly the verification of quantum algorithms. We have seen that verification can be done in many ways, and have contributed to the approach which attacks this problem using decision diagrams. We now reflect on our findings by revisiting the research questions posed in Chapter 1. Finally, we conclude by posing questions for future research.

8.1 Research questions, revisited

Quantum algorithms can be verified automatically using several methods, which we briefly surveyed in Section 2.4, and using a variety of data structures, which we examined in Chapter 5. The principal bottleneck in these methods is the large amount of computer memory required to store the state vectors of the quantum states that are encountered during simulation (or to store the unitary matrix of a quantum circuit). To this end, these data structures are used to losslessly compress both these states vectors and the unitary matrices of quantum circuits. We have argued that, therefore, powerful data structures are the key to effective methods for verification and indeed for quantum algorithm analysis more broadly. Concretely, a bottleneck for the approaches we studied was that decision diagrams cannot efficiently analyze stabilizer circuits, an important class of quantum circuits. Specifically, existing decision diagrams require exponential amounts of space and time for this use case (Theorem 3.2). In light of this significant limitation, we asked:

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

To combat this memory bottleneck, in Chapter 3 we introduced a new data structure for the analysis and verification of quantum circuits: the Local Invertible Map Decision Diagram (LIMDD), which unites the strengths of existing decision diagram-based approaches and the stabilizer formalism. We compared the LIMDD to several existing data structures in Chapter 3, 4 and 5 and found that LIMDDs are exponentially faster (in a qualitative sense: they are more *rapid*) and more succinct than QMDDs, more expressive than the stabilizer formalism; and incomparable to matrix product states, restricted Boltzmann Machines and the extended stabilizer formalism.* These results were obtained in a mathematically rigorous way: we provided examples of quantum states which LIMDDs can represent efficiently but which require exponentially scaling resources when using the data structures above and vice versa. The Venn diagram in Figure 8.1 summarizes these results.

We showed empirically that LIMDDs can analyze an important quantum subroutine called the Quantum Fourier Transform, in Chapter 4. To this end, we simulated the circuits using random stabilizer states as the inputs. Due to their better asymptotic scaling, LIMDDs outperform QMDDs on this task: they are faster on circuits that contain 19 qubits or more, and are about 5 times faster on the largest circuit that was tested.

In the introduction, we sketched several shortcomings of the literature on knowledge compilation with regards to data structures that are used for quantum algorithm analysis. Notably, existing data structures in widespread use in this field had not been compared on their succinctness and could not be compared on their rapidity; and the tradeoffs between the tractability of key operations had not been systematically compared across existing data structures. Therefore, we asked:

> **Research question 2.** *How can we analytically compare the relative strengths of data structures which represent quantum states?*

We answered Research question 2 by giving a quantum knowledge compilation map

---

*Specifically, we compared LIMDDs to a specific instantiation of the extended stabilizer formalism which captures the way this formalism is often used in practice; for details see Sec. 3.3.4.

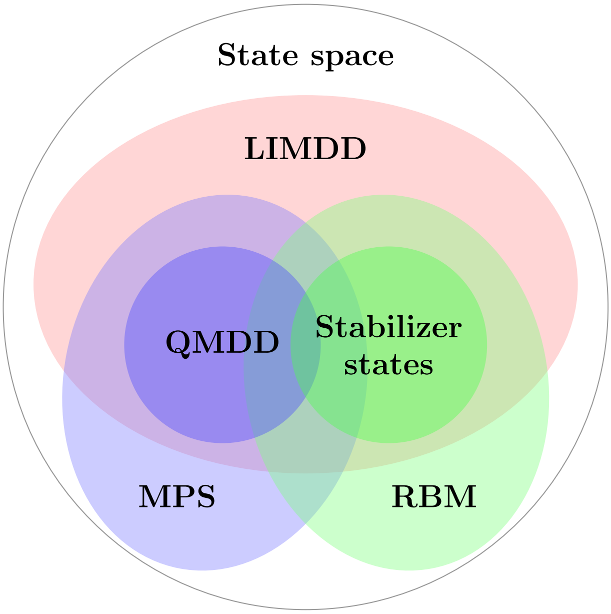


Figure 8.1: A Venn diagram showing the expressive power of almost all data structures studied in this thesis. The circle of a data structure (e.g., the pink circle labeled LIMDD) can be interpreted as either (i) the set of quantum states that can be represented in polynomial space using that data structure, thus depicting the *succinctness relations*; or (ii) the set of tasks that can be solved in polynomial time using this data structure, thus depicting the *rapidity relations*.

in the style of Darwiche and Marquis [97] and Fargier et al. [113] in Chapter 5 in which we mapped the succinctness, tractability and rapidity of many data structures used in quantum algorithm analysis: ADDs, QMDDs, LIMDDs, matrix product states, restricted Boltzmann machines and the explicit state vector representation. The Venn diagram in Figure 8.1 summarizes these results. Notably, we found that LIMDDs and matrix product states are more rapid than QMDDs.

To this end, we extended the definition of rapidity to also cover the case of non-canonical data structures. We then gave a simple sufficient condition for when one (non-canonical) data structure is more rapid than another. This sufficient condition allowed us to establish almost all rapidity relations between the data structures we studied. We argued that rapidity is an especially useful metric when choosing which data structure to use for a given task because, contrary to the tractability criterion, it directly compares the absolute amounts of time two data structures require for a given operation, correctly taking into account the fact that they perform the operation on instances of different sizes due to their different succinctness. Lai et al. [194] formulate this same idea: "[rapidity] reflects the idea that exponential operations may be preferable if the language can be exponentially more succinct."

By contrast, the criterion of tractability was found to be less informative because it appears to "punish" data structures for adding functionality which increases their succinctness, even when canonicity is preserved and this additional functionality does not increase the absolute time required for a given operation. Let us briefly give two examples of this phenomenon to illustrate this subtle point. First, applying a Hadamard gate to a quantum state is tractable in the state vector representation but is intractable for QMDDs and LIMDDs. Second, computing the fidelity of two quantum states is tractable for QMDDs but intractable for LIMDDs. Both of these results are counterintuitive, since both the step from the state vector representation to QMDDs, and from QMDDs to LIMDDs, are steps which add functionality to a data structure, which improve their succinctness and which do not increase the runtimes of any operations (up to polynomial factors). The criterion of rapidity, on the other hand, yields the expected result that LIMDDs are more rapid than QMDDs, which in turn are more rapid than the state vector representation, for all these operations.

We further answered Research question 2 in Chapter 3 by analytically comparing LIMDDs to matrix product states and the extended stabilizer formalism (ESF). To this end, we gave an example of a quantum circuit which LIMDDs can analyze efficiently, whereas we describe a specific instantiation of the ESF which requires exponential time for this same task, conditional on the exponential time hypothesis.

In the future, we hope that researchers will investigate the rapidity of the data structures they analyze, both when introducing new data structures and when making knowledge compilation maps. More broadly, we hope that rapidity, rather than succinctness or tractability, will become the primary design objective when designing new data structures.

We remarked that very few decision diagrams have been adapted to the quantum setting and asked in which cases it might be lucrative to do so:

> **Research question 3.** *Which classical decision diagrams might be effective for the analysis of quantum algorithms, if they were suitably adapted?*

To answer this research question, in Chapter 6 and 7, we investigated two DDs which we find promising candidates for future development in quantum settings: the DSDBDD and the SDD, respectively. We now treat these two DDs in turn.

We found that the DSDBDD is exponentially more succinct than the BDD due to its superior ability to recognize structure in a Boolean function. We comment more on how to extend this data structure to the quantum setting in Sec. 8.3.2.

We are the first to apply SDDs to model checking. In Chapter 7, we show the empirical results on a large standard model checking benchmark set. Recall that an SDD extends BDD by using a variable tree, which generalizes the BDD's notion of a variable order. A vtree consists of a full binary tree in addition to a variable order. Both these components determine the size of the SDD and the speed with which manipulation operations can be carried out, so a tool which uses an SDD needs to make good choices for both components in order to obtain good performance. To choose the vtree, we proposed the first heuristics for this purpose and found that in general it helps to group related variables together. We found that SDDs using these heuristics often outperform BDDs on memory consumption, and sometimes also on time, if its vtree is chosen well. For example: noting that an integer program variable consists of multiple bits, for every such integer variable, the vtree should contain a subtree which contains only the bits of this integer variable. The resulting software package integrates SDDs into the publicly available, open source model checker LTSmin [210]. To extend the SDD to the quantum domain, a promising start may be to draw inspiration from the Probabilistic SDD [180] and Tree Tensor Networks [236, 290].

8.2 Discussion

Unless P = BQP,[†] any classical simulation of quantum systems will have to make some tradeoff between speed, memory consumption, and accuracy, or has to specialize in a restricted set of circuits. The LIMDDs presented in this work represent a novel point in this optimization space, but they necessarily make some tradeoff compared to other methods. Therefore, we briefly but critically reflect on the limitations of LIMDDs as a data structure, and on the work in this thesis more broadly.

Compared to QMDDs, LIMDDs have an $\mathcal{O}(n^3)$ computational overhead when perform-

---

[†] It is widely believed that P $\neq$ BQP. See [20] for definitions of these complexity classes.

ing almost any manipulation operation. The purpose of the subroutines that cause this overhead is to allow for exponentially greater compression, i.e., to allow for less memory consumption; indeed, this compression allows LIMDDs to be faster in those cases where the workload saved due to compression outweighs the overhead which facilitated this compression. However, this overhead may become a liability in cases where LIMDDs obtain no significant compaction. Fortunately, in Chapter 4 we have seen that this worst case scenario does not necessarily materialize, as LIMDDs outperform QMDDs when analyzing the quantum Fourier transform even though their size is not much smaller. This can be explained by the fact that most of the overhead is due to Gaussian elimination of the stabilizer group stored with each node, but in this case these groups are almost all empty, so there is almost no work to be done.

Numerical accuracy and numerical stability are problems for any real-valued decision diagram [156, 157] and LIMDDs are no exception. We have not investigated to what extent these problems affect LIMDDs and leave this important question to future work. An optimistic hypothesis is that one should expect to encounter these problems to a lesser extent for LIMDDs than for QMDDs because, all other things equal, LIMDDs contain fewer floating point numbers than QMDDs do, simply because they are smaller.

We concluded Chapter 4 by reflecting on the methodology of circuit equivalence testing via simulation of random states. We remarked that the theoretical guarantees given by stabilizer states are good, but not perfect (specifically, the expected fidelity is optimal in the black-box setting, but the probability that a stabilizer state is a counterexample is less than that of a Haar-random state). Therefore, if we wish to obtain better guarantees, we will need to draw the random inputs from a different set of quantum states. In choosing this set, there is a tension between a set which is *random enough* to yield good theoretical guarantees, yet *structured enough* that computational methods are effective. Concretely, using a random stabilizer state to analyze a circuit U is equivalent to effecting a change of basis and analyzing instead the matrix $C^\dagger UC$, where C is a Clifford circuit that produces the random stabilizer state. Since C is by design a random unitary, one may therefore expect that, all other things equal, the matrix $C^\dagger UC$ will possess less structure than U, and that computational methods will therefore have a harder time analyzing it.

In Chapter 5, we mapped the tractability of several elementary operations for many data structures. However, the tractability of elementary operations is not always the most informative data point about a given data structure. Instead, it is often more useful to know whether a data structure can efficiently analyze a given family of quantum circuits – quantum circuits, of course, being successive applications of elementary operations (i.e., gates). Making a knowledge compilation map always involves making such a sacrifice: on one hand, we abstract away from concrete sets of use cases (such as specific families of quantum circuits, in our case), which diminishes the immediate applicability of the results; on the other hand, by examining which elementary operations are efficiently supported by a data structure, one ostensibly gains insights which are more likely to be general enough to carry over from one use case to another. We say more about future work which might diminish the chasm between these opposing goals in Research challenge 11.

8.3 Future work

The work in this thesis does not close the book on quantum algorithm analysis or decision diagram research – quite the contrary: we have raised more questions than we have answered. We list here several open problems which we find promising avenues for future research. We first lay out future work for specific decision diagrams and then ask several more general questions.

8.3.1 Future work on LIMDDs.

There are several avenues for further development of LIMDDs. Broadly speaking, we see three categories: (i) LIMDDs should be deployed in practical and easy-to-use software applications for quantum simulation and verification; (ii) the existing LIMDD implementations can be improved and extended; and (iii) there are several analytical questions about the expressive power of LIMDDs, which can shed light on which LIMDD extensions are most promising.

Research challenge 1. *Facilitate the deployment of LIMDDs.*

Although we have implemented LIMDDs and although the software package is versatile and easy to use, the package is currently more or less a stand-alone application. However, a software package will only start to deliver value when it is integrated into a toolchain or an IDE in a way that makes it accessible and easy to use for the end user – in this case, programmers of quantum computers (as noted also by, e.g., Raed el Aoun et al. [269]).

Research challenge 2. *Develop LIMDDs for classical applications.*

We are enthousiastic about the potential of LIMDDs in the classical domain. To this end, we must redefine what it means for two Boolean functions to be isomorphic. For example, we can say that two Boolean functions $f, g\colon \{0,1\}^n \to \{0,1\}$ are Pauli-isomorphic if there exist Boolean values $z_1, \ldots, z_n, a_1, \ldots, a_n, b \in \{0,1\}$ such that

$$f(x_1, \ldots, x_n) = b \oplus x_1 z_1 \oplus \cdots \oplus x_n z_n \oplus g(x_1 \oplus a_1, \ldots, x_n \oplus a_n) \quad \text{for all } x \in \{0,1\}^n \tag{8.1}$$

Thus, two functions are classically Pauli-isomorphic if they are equal modulo (i) complementation of the variables combined with (ii) an affine linear function. An encouraging first result is that we can immediately obtain an exponential separation between classical Pauli-LIMDD versus OBDD because the separation between $\langle X \rangle$-LIMDD and QMDD for coset states (Corollary A.1) carries over to this context. Namely, in the classical domain, the equivalent of a “coset state” is a set of solutions to a system of linear equations over $\mathbb{F}_2$.

Let us name two appealing properties of exploring the classical domain. First, many DDs seem to falter at “two-dimensional” problems (e.g., [318]), i.e., problems in which the variables are arranged in a grid, whereas LIMDDs have been shown to solve at least one such problem, namely it can efficiently represent and manipulate graph states. Second, numerical instability (resulting from inaccurate floating-point arithmetic) will likely not be an issue. Therefore, we can efficiently and reliably apply dynamic variable reordering to these diagrams, as opposed to the quantum case [156].

Research challenge 3. *Can DDs with other architectures than a variable order benefit from LIMs?*

Although the landscape of decision diagrams is rich and diverse, the underlying architecture of every proposed decision diagram can (to the best of the author’s knowledge) be classified as one of the following three (see also Table 2.2, which lists examples decision diagrams using the various architectures):

1. architectures based on a **variable order** (these constitute the vast majority of implementations, including BDD, [67] ZDD, [229] DSDBDD, [46, 264] TBDD, [328] KFBDD, [103] LIMDD, [337] QMDD, [227, 374] AADD, [281]...);
2. architectures based on a **variable tree** (these include the SDD [96] and its extensions: the Tagged SDD, [111] ZSDD, [245] VS-SDD [235] and probabilistic SDD [180]);
3. architectures based on a **variable decision diagram** (these include the graph-based BDD [129] and Partitioned ROBDD [237]).

In our work, we have explored only architectures based on a variable order. We are excited to note that, by following the simple guiding principle that isomorphic nodes should be merged, a LIMDD-like structure can be implemented on any of these architectures. More generally, we can take any one of the decision diagrams named above as a starting point and augment it with LIMs to obtain a new, canonical diagram. This way we may, in principle, obtain, e.g., (i) a combination of CCDD [195] with classical LIMDD; or (ii) a zero-suppressed LIMDD (by skipping nodes of the form $|\varphi\rangle = |0\rangle |\varphi_0\rangle$); or any other combination. Of course, this is easier said than done: we note that it is easy to *define* a decision diagram, and much harder to *realize* it in software, especially if its performance needs to compete with the state-of-the-art.

> **Research challenge 4.** *Analyze and implement G-LIMDDs for various choices of G.*

In our work, we have worked primarily with $G = \textsc{Pauli}$, but there are many interesting choices that lead to more succinct and more rapid G-LIMDDs . For example, the choice $G = \langle X, T\rangle$ allows us to apply T gates in constant time and controlled-T gates in polynomial time. Alternatively, by taking the cyclic group $G = \langle R(n)\rangle$ generated by the matrix $R(n) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi 2^{-n}} \end{bmatrix}$ we can represent the Quantum Fourier Transform (QFT) in only $\mathcal{O}(n^4)$ space. This is a big improvement compared to QMDD and Pauli-LIMDD, for which the QFT is a worse-case situation (namely, these diagrams require size $\Omega(2^n)$). Giving LIMDDs slightly more power still, the group $G = \langle X, R(n)\rangle$ allows us to efficiently represent the QFT, all Clifford circuits, all stabilizer states and the so-called XS-stabilizer states studied by Ni et al. [239]. More ambitiously, we can consider multi-qubit groups, e.g., the group $G = CZ$ generated by all controlled-Z gates. This group allows us to represent in polynomial space (i) hypergraph states and (ii) all circuits composed only of Z gates with zero, one, or two control qubits. We may even combine this group with, e.g., the Pauli group. More generally, whenever we have efficient algorithms constructing G-LIMDDs and H-LIMDDs for two groups G and H, we may consider constructing the J-LIMDD with $J = \langle G, H\rangle$. The LIMDD using the group of controlled-Z gates, proposed above, also has a natural classical analogue: two Boolean functions f, g are "Controlled-Z-isomorphic" if they are equal modulo a degree-2 multilinear polynomial q over $\mathbb{F}_2$, i.e., if $f(x) = q(x) \oplus g(x)$.

In the spirit of exploring the opportunities afforded by choosing a group G, it is natural to ask the following:

> **Research challenge 5.** *Prove succinctness and rapidity separations between G, H-LIMDDs for groups G, H.*

We have seen that the capability of a G-LIMDD to represent and manipulate quantum circuits is determined by the choice of the group G. For example, PAULI-LIMDDs can represent all stabilizer states, including coset states, but $\langle Z\rangle$-LIMDDs cannot efficiently represent coset states: PAULI-LIMDDs are therefore strictly more succinct and more rapid than $\langle Z\rangle$-LIMDDs. It is always true that, if $G \subseteq H$, then H-LIMDDs are at least as succinct as G-LIMDDs . Is this separation always strict when $G \subsetneq H$? Is there always a separation when $G \setminus H$ is non-empty? These questions would be answered by a map displaying how the choice of G affects the succinctness of the resulting G-LIMDD . On one hand, such a map may in principle be convoluted, since G may be chosen to be any one of the uncountably infinitely many groups of invertible matrices, i.e., subgroups of $GL(2)$. On the other hand, settling the following conjecture in the affirmative would fully resolve this question for succinctness (although not for rapidity).

Conjecture 8.1. If G and H are single-qubit groups, and if $G \setminus H$ is non-empty, then there are states with polynomial-size G-LIMDD but which have only exponential-size H-LIMDD. In fact, there is a Tower-G-LIMDD with this property.

> **Research challenge 6.** *When is measurement NP-hard in LIMDDs?*

In this work, we have given fast algorithms for measurement in G-LIMDDs for certain choices of G, namely for $G \subseteq$ Pauli. This is important to allow fast simulation of quantum computation. These same algorithms would work equally well for any G-LIMDD where G contains only unitary matrices. However, if G contains non-unitary matrices, the algorithm that we presented must be modified, and the result is no longer polynomial-time. To see why, recall that the squared norm of an edge $\xrightarrow{\lambda P}\textcircled{v}$ is $|\lambda^\dagger\lambda| \cdot |\langle v| P^\dagger P |v\rangle|$. Therefore, if G is unitary, then $P \in G$ is unitary, so $P^\dagger P = \mathbb{I}$, so we may recurse to compute the squared norm of node v. By contrast, a non-unitary P will not effect this cancellation, and the operator $P^\dagger P$ must be propagated to the recursive call; consequently, the runtime of the algorithm may become exponential, even with dynamic programming. It is natural to wonder whether this exponential running time is necessary, and if so, for which non-unitary groups G? To this end, we formulate the following conjecture for one such group.

Conjecture 8.2. Measurement in G-LIMDD is NP-Hard for $G = \{[\begin{smallmatrix}1 & 0\\ 0 & x\end{smallmatrix}] \mid x \in \mathbb{R}_{\neq 0}\}$. Specifically, given such a LIMDD for a quantum state $|\varphi\rangle$, it is NP-Hard to compute the probability of the outcome of a measurement of the first qubit. This remains true even when the LIMDD is a Tower.

8.3.2 Future work on DSDBDDs

Research challenge 7. *Is DSDBDD more rapid than BDD?*

We have shown that DSDBDD is more succinct than BDD. Bertacco, Damiano and Plazo gave an efficient way to convert a BDD to a DSDBDD [46, 264]. This suggests that one may apply the sufficient condition we formulated in Chapter 5 to show that DSDBDD is more rapid than BDD.

Research challenge 8. *Can we develop a Quantum DSDBDD?*

The definition of DSDBDD (Definition 6.2) effects a diagram which encodes a Boolean function (i.e, of the form $f\colon \{0,1\}^n \to \{0,1\}$), not a pseudo-Boolean function. It is not self-evident how a composition $k \circ g_1, \ldots, g_m$ of a kernel k with factors $g_1, \ldots, g_m$ should be defined for pseudo-Boolean functions unless the range of the factors g_j is reconciled with the domain of k. We make a suggestion here and defer other necessary considerations to future research, e.g., finding reduction rules to make the diagram canonical, and finding algorithms to keep the diagram reduced, noting that we believe that such needs can be met. For any pseudoboolean function $k\colon \{0,1\}^n \to \mathbb{C}$, let $\overline{k}\colon \mathbb{C}^n \to \mathbb{C}$ be the unique multilinear extension of k. In more detail, any pseudoboolean function k has a unique expression as a weighted sum of monomials:

$$k(x_1, \ldots, x_n) = \sum_{S \subseteq \{x_1, \ldots, x_n\}} \alpha_S \prod_{x \in S} x \qquad \text{with } \alpha_S \in \mathbb{C} \tag{8.2}$$

This expression can be interpreted as a multilinear polynomial $\overline{k}$ over $\mathbb{C}$; this polynomial $\overline{k}$ is called the (unique) multilinear extension of k. Using the concept of multilinear extensions, we can define the composition of pseudoboolean functions as follows. Let $k, g_1, \ldots, g_m$ be pseudoboolean functions. Then the composition $h = k \circ (g_1, \ldots, g_m)$ is defined as the following pseudoboolean function h,

$$h(\vec{x}_1, \ldots, \vec{x}_m) = \overline{k}(g_1(\vec{x}_1), \ldots, g_m(\vec{x}_m)) \tag{8.3}$$

Thus, the composition of pseudoboolean functions yields another pseudoboolean function. For example,[‡] using the kernel $k(x_1, x_2) = x_1 \cdot x_2$, we get that $k \circ (|\varphi_1\rangle, |\varphi_2\rangle) = |\varphi_1\rangle \otimes |\varphi_2\rangle$. That is, we can easily recover the familiar tensor product by considering compositions of pseudoboolean functions. Which other familiar constructions might this tool elucidate?

8.3.3 Future work on SDDs

Research challenge 9. *Can SDDs be augmented with disjoint support decompositions?*

We have seen that extending BDDs with DSDs significantly improves their succinctness (and perhaps their rapidity). Is the same true of SDDs? A good first step is to show that DSDs make SDDs more succinct (the example we provided in Chapter 6 likely suffices for this purpose). The more difficult second step is finding an algorithm which takes the SDD node of a Boolean function f and finds a decomposition $f = k \circ (g_1, \ldots, g_m)$, if one exists.

[‡] Some abuse of language happens in this example: we write $|\varphi\rangle$ where we mean the amplitude function of that state.

Research challenge 10. *Is conjunction of SDDs tractable?*

Van den Broeck and Choi raised this question [322]. They showed that the answer is *no* if the output vtree must be the same as the input vtrees. So any polynomial-time SDD conjunction algorithm, if it exists, must sometimes output an SDD with a different vtree than the input SDDs. A weaker question is to ask whether there always *exists* a polynomial-size SDD representing the conjunction of two DDs, regardless of the question of how to find it algorithmically.

8.3.4 Future work on quantum knowledge compilation

We sketch two avenues for future work in the area of knowledge compilation.

First, in our knowledge compilation map, we map the succinctness, tractability and rapidity of many popular quantum data structures. However, we omit the popular tensor network data structure. There is a rich literature on tensor networks employing various topologies, e.g., arranging the tensors on a line, on a grid, and in other ways [110, 248]. It is difficult to place these many variations on the knowledge compilation map in a meaningful way because the manipulation operations are often tractable, and indeed trivial (e.g., one "applies" an elementary gate to a tensor network in $\mathcal{O}(1)$ time by simply appending it to the open indices of an existing tensor network), whereas the queries are often intractable (they are #P-complete in many cases). Therefore, the many variations of tensor networks each look identical from the point of view of tractability. Moreover, if there are no (efficient) transformations which turn one type of tensor network into another, then rapidity results will be difficult to obtain. We leave it to future research to devise methodologies which allow to compare these data structures.

Second, in both our and Lai et al.'s [194] definition of rapidity, we require that a data structure be more rapid *on all instances*. However, it is often interesting to look at a restricted class of use cases, for example, one might be interested only in stabilizer circuits. Therefore, it is often natural tot ask whether one data structure is more rapid than another *on a particular family of inputs.*

> **Research challenge 11.** *Formulate and investigate a notion of rapidity restricted to a given family of inputs.*

For example, we have shown that restricted Boltzmann machines (RMBs) are incomparable to QMDDs in terms of rapidity, but we may expect that restricted Boltzmann machines are more rapid than QMDDs when restricted to analyzing stabilizer circuits. This allows for a more fine-grained comparison between two data structures which ordinarily would be incomparable in terms of rapidity. This may be an avenue for approaching the question raised above, where it is difficult to compare different versions of tensor networks: each may have their own strengths when analyzing particular families of circuits and this may be illuminated by a suitable notion of rapidity.

8.3.5 Future work on verification and decision diagrams

In this thesis, we have attacked the problem of quantum circuit equivalence checking. But quantum algorithm verification is broader than merely equivalence checking; therefore, we ask:

> **Research challenge 12.** *How can we do quantum algorithm verification beyond circuit equivalence checking?*

We see three ways in which algorithm verification is broader than merely circuit equivalence checking, which present three avenues for research:

- **Use a specification which is not another quantum circuit.**
  We have seen (in Section 2.4) that there are several temporal logics that are tailor-made for quantum applications, such as QCTL, QLTL, and quantum Hoare logic [30, 79, 100, 216, 217, 355, 359, 363, 366, 367]. Several model checking tools for such quantum temporal logics already exist (e.g., [116, 128, 161, 205]). There are good reasons to prefer using a temporal logic over circuit equivalence checking in certain situations: in such a logic, a user can specify the intended behaviour of an algorithm in a more natural and more fine-grained way than they can in circuit equivalence checking; e.g., in QCTL, it is possible to assert that two given qubits eventually become entangled [30]. Given that DD-based

techniques have been fruitful in the contexts of (i) classical model checking using similar temporal logics and of (ii) quantum circuit equivalence checking, we are optimistic that these same techniques can help bring the theory of quantum model checking with temporal logics into practice. To the best of our knowledge, no DD-based approach has been used in this context.

- **Verify an *algorithm* rather than a *circuit*.**
  An *algorithm* is often understood to receive an input of unbounded length, as opposed to a *circuit*, which takes an input of fixed length. For example, looking back at Chapter 4, even if we were to verify that an implementation of QFT is correct for $n = 2 \dots 24$ qubits, this would not guarantee correctness for $n = 25$. Therefore, the more important and more daunting task is to verify the correctness of the (classical) procedure which generates the circuits. The work of Amy et al. [13, 15], Hietala et al. [155] and Peng et al. [261] is in this direction, for example.

- **Equivalence checking of circuits with different numbers of ancilla qubits.**
  Two circuits may be understood to be equivalent, even if they use different numbers of ancilla qubits (this is akin to comparing two classical algorithms which use different amounts of memory). For example, multiple authors have cut down the number of ancilla qubits required for Shor's algorithm [35,309], but two such implementations can still be considered equivalent if the measurements at the end of the circuits produce the same outcomes. Therefore, ideally we would ask about the equivalence of two superoperators, rather than two unitary matrices. For example, Ardeshir-Larijani take this approach [17]; Hong et al. use DDs to check the (approximate) equivalence of two superoperators corresponding to two implementations [163]; and similarly Grurl, Fuß and Wille [141,143] use QMDDs to represent density matrices. Therefore, although steps have been taken in this direction, more research is called for, since this is the more natural use case in many applications. Moreover, this research goal could be a stepping stone to address the two research goals above.

Lastly, we ask how to compare decision diagrams in terms of the problems they can solve, rather than the languages they represent.

Research challenge 13. *Characterize the problems that can be solved with decision diagrams.*

This question is best illustrated by two motivating examples:

1. A BDD can succinctly represent the set of satisfying assignments of any CNF whose incidence graph has bounded treewidth [273]. In fact, such a BDD can be constructed in polynomial time, and (therefore) BDDs can efficiently decide its satisfiability. However, BDDs cannot succinctly represent the set $F \subset \mathsf{CNF}$ of satisfiable CNF formulas having bounded treewidth.
2. Say that a quantum circuit *accepts* if the probability of obtaining a 1 in the final measurement is greater than 1/2; otherwise it *rejects*. Then, since PAULI-LIMDDs can efficiently simulate Clifford circuits, including computing measurement outcome probabilities, they can efficiently decide whether a given Clifford circuit accepts or rejects. However, PAULI-LIMDDs cannot succinctly represent the set of accepting Clifford circuits.

We see that there is a difference between the problems a DD can *solve*, and the languages a DD can *represent*. The set of languages represented by a DD is traditionally denoted by a complexity class; e.g., the complexity class BDD contains all decision problems $L \subseteq \{0,1\}^*$ such that for each $n \geq 0$, the set $L_n = L \cap \{0,1\}^n$ has a polynomial-size BDD. However, the two examples above illustrate that it may be at least as interesting to consider the complexity class containing all decision problems solved by a given DD. The latter example in particular is remarkable because the task of simulating a Clifford circuit is $\oplus$L-complete[§] [3] (For our purposes, it suffices to say that $\oplus$L is a complexity class containing problems of considerable computational difficulty).

Therefore the question arises: given a decision diagram, which complexity class is characterized by the problems that can be efficiently solved using that decision diagram, for example, using a BDD, or a PAULI-LIMDD? We immediately obtain the following upper bound. Namely, by definition, if some DD $\mathcal{D}$ efficiently solves a decision problem

---

[§] Recall that L is the complexity class Logspace, of all problems solvable by a Turing Machine using $\mathcal{O}(\log n)$ space. Here $\oplus$L is the complexity class of problems solvable by a non-deterministic TM using $\mathcal{O}(\log n)$ space, and where the TM is defined to accept an input iff there is an odd number of accepting nondeterministic paths.

$L \subseteq \{0,1\}^*$, then using this decision diagram constitutes a polynomial-time algorithm which solves L, so L is in P. Consequently, if $\mathcal{C}_{\mathcal{D}}$ is the complexity class of problems efficiently solved using this DD, then $\mathcal{C}_{\mathcal{D}} \subseteq \mathrm{P}$. Can we design a decision diagram which achieves $\mathcal{C}_{\mathcal{D}} = \mathrm{P}$? Can we design decision diagrams which achieve other notable classes $\mathcal{C}_{\mathcal{D}}$? We leave it up to future research to more precisely define what it means for a decision diagram to “solve” a certain decision problem L.

Appendix A

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, Theorem 3.2 and Corollary A.1). On the other hand, in App. B, we will show that these states can be represented using only $\mathcal{O}(n)$ 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 $V_G = \{v_1, ..., v_n\}$ 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

$$|G\rangle = \sum_{\vec{x}\in\{0,1\}^n} (-1)^{f_G(\vec{x})} |\vec{x}\rangle \tag{A.1}$$

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 $\vec{a}$:

$$f_{\vec{a}}(x_{k+1},\ldots,x_n) := f(a_1,\ldots,a_k,x_{k+1},\ldots,x_n) \tag{A.2}$$

We also say that $f_{\vec{a}}$ is a subfunction of f of *order* $|\vec{a}| = k$.

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

Definition A.1 (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 A.2 (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 $S, T \subset V$ 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 Theorem 3.2.

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

Proof. 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 $S = \{v_1, v_2, \ldots, v_{\frac{1}{2}n^2}\} \subset V$ be the first $\frac{1}{2}n^2$ vertices in this order.

The proof proceeds broadly as follows. First, in Lemma A.1, we show that any $(S,\overline{S})$-strong matching M effects $2^{|M|}$ different subfunctions of f_G. Second, Lemma A.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 Theorem 3.2 by the fact that a QMDD for the cluster state G has a node per unique subfunction of the function f_G. Figure A.1 illustrates this setup for the 5×5 lattice.

Lemma A.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 $\chi(x_1, ..., x_n)$ for the indicator function for vertices: $\chi(x_1, ..., x_n) := \{v_i \mid x_i = 1, i \in [n]\}$. Choose two different subsets $A, B \subseteq S_M$ and let $\vec{a} = \chi^{-1}(A)$ and $\vec{b} = \chi^{-1}(B)$ be the corresponding length-$|S|$ bit strings. These two strings induce the two subfunctions $f_{G,\vec{a}}$ and $f_{G,\vec{b}}$. We will show that these subfunctions differ in at least one point.

First, if $f_{G,\vec{a}}(0, \dots, 0) \neq f_{G,\vec{b}}(0, \dots, 0)$, 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

$$f_{G,\vec{a}}(0, \dots, 0, x_t = 0, 0, \dots, 0) \neq f_{G,\vec{a}}(0, \dots, 0, x_t = 1, 0, \dots, 0) \tag{A.3}$$

$$f_{G,\vec{b}}(0, \dots, 0, x_t = 0, 0, \dots, 0) = f_{G,\vec{b}}(0, \dots, 0, x_t = 1, 0, \dots, 0) \tag{A.4}$$

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 A.2. Let $S = \{v_1, \dots, v_{\frac{1}{2}n^2}\}$ 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 [204]. Each vertex of the boundary of S has degree at most 4. It follows that there is a set of $\lfloor \frac{1}{4}|B_S| \rfloor$ vertices which share no neighbors. In particular, there is a set of $\lfloor \frac{1}{4}|B_S| \rfloor \geq \lfloor \frac{1}{12}n \rfloor$ 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 $n/3$ nodes, which yields a strong matching of at least $\lfloor \frac{1}{12}n \rfloor$ edges, which shows that f_G has at least $2^{\lfloor \frac{1}{12}n \rfloor}$ 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 Ďuriš 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.

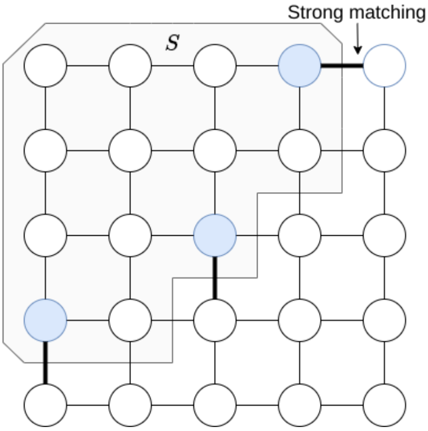


Figure A.1: The 5×5 lattice, partitioned in a vertex set S and its complement $\overline{S}$. A strong matching between S and $\overline{S}$ is indicated by thick black edges. The nodes in S are highlighted.

Theorem A.1 (Ďuriš et al. [108]). The characteristic function $f_V : \{0,1\}^n \to \{0,1\}$ 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 A.1. 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 $f\colon \{0,1\}^n \to \{0,1\}$. 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 $f\colon \{0,1\}^n \to \mathbb{C}$, 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 $f_{V,a} \neq \lambda f_{B,b}$.

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 Theorem A.1 with high probability the BDD representing f_V has exponentially many nodes, so does the QMDD representing $|V\rangle$. $\square$

Appendix B

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 $\langle \textsc{Pauli}\rangle$-Tower-LIMDDs correspond to graph states, coset states, and stabilizer states, respectively, in Theorem B.1, Theorem B.2 and Theorem 3.1 below. Definition 3.5 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 Definition 3.2 which allows parameterization of the LIM G. They only rely on the Tower LIMDD in Definition 3.3.

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 graph G gives rise to a graph state $|\varphi_G\rangle$ represented by a LIMDD. Let $P = P_{k-1} \otimes \cdots \otimes P_1$ be the label on the high edge out of the LIMDD node at level k. Then G contains an edge (v_k, v_j) 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_v 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. Definition 3.3 gives an exact definition.

Theorem B.1 (Graph states are $\langle Z\rangle$-Tower-LIMDDs)**.** Let $n \geq 1$. Denote by $\mathcal{G}_n$ the set of n-qubit graph states and write $\mathcal{Z}_n$ for the set of n-qubit quantum states which are represented by $\langle Z\rangle$-Tower-LIMDDs a defined in Definition 3.3, i.e, a tower with low-edge-labels $\mathbb{I}$ and high-edge labels $\lambda \bigotimes_j P_j$ with $P_j \in \{\mathbb{I}, Z\}$ and $\lambda = 1$, except for the root edge where $\lambda \in \mathbb{C} \setminus \{0\}$. Then $\mathcal{G}_n = \mathcal{Z}_n$.

Proof. We establish $\mathcal{G}_n \subseteq \mathcal{Z}_n$ by providing a procedure to convert any graph state in $\mathcal{G}_n$ to a $\langle Z\rangle$-Tower-LIMDD in $\mathcal{Z}_n$. See Figure B.1 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 Equation A.1), 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 Figure B.1(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 Equation A.1:

$$|G\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle \otimes |G_{1..n}\rangle + |1\rangle \otimes \underbrace{\left[\bigotimes_{(n+1,j)\in E} Z_j\right]}_{\text{Isomorphism B}} |G_{1..n}\rangle\right) \tag{B.1}$$

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 $\in \mathcal{Z}_n$. Now we construct a $\langle Z\rangle$-Tower-LIMDD for $|G\rangle$ as follows. The root node has two outgoing edges, both going to the node representing $|G_{1..n}\rangle$.

The node's low edge has label $\mathbb{I}$, and the node's high edge has label B, as follows,

$$B = \bigotimes_{(n+1,j)\in E} Z_j \tag{B.2}$$

Thus the root node represents the state $|0\rangle |G_{1..n}\rangle + |1\rangle B |G_{1..n}\rangle$, satisfying Equation B.1.

To prove $\mathcal{Z}_n \subseteq \mathcal{G}_n$, 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 $\mathbb{I}$.

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 $\mathbb{I}$, and a high edge labeled $P_k \otimes \cdots \otimes P_1$, with $P_j = Z^{a_j}$ for $j = 1 \dots k$. Here by Z^{a_j} we mean $Z^0 = \mathbb{I}$ 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 $j = 1 \dots k$. The state represented by this node is of the form given in Equation B.1, so it represents a graph state.

A simple counting argument based on the above construction shows that $|\mathcal{Z}_n| = |\mathcal{G}_n| = 2^{\binom{n}{2}}$, 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 $\mathbb{I}$ or Z. $\square$

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

Theorem B.2 (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 $\mathcal{X}_n$ for the set of n-qubit quantum states which are represented by $\langle X\rangle$-Tower-LIMDDs as per Definition 3.3, i.e., a tower with low edge labels $\mathbb{I}$ and high edge labels $\lambda \bigotimes_j P_j$ with $P_j \in \{\mathbb{I}, X\}$ and $\lambda \in \{0, 1\}$, except for the root edge where $\lambda \in \mathbb{C} \setminus \{0\}$. Then $\mathcal{V}_n = \mathcal{X}_n$.

Proof. We first prove $\mathcal{V}_n \subseteq \mathcal{X}_n$ 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

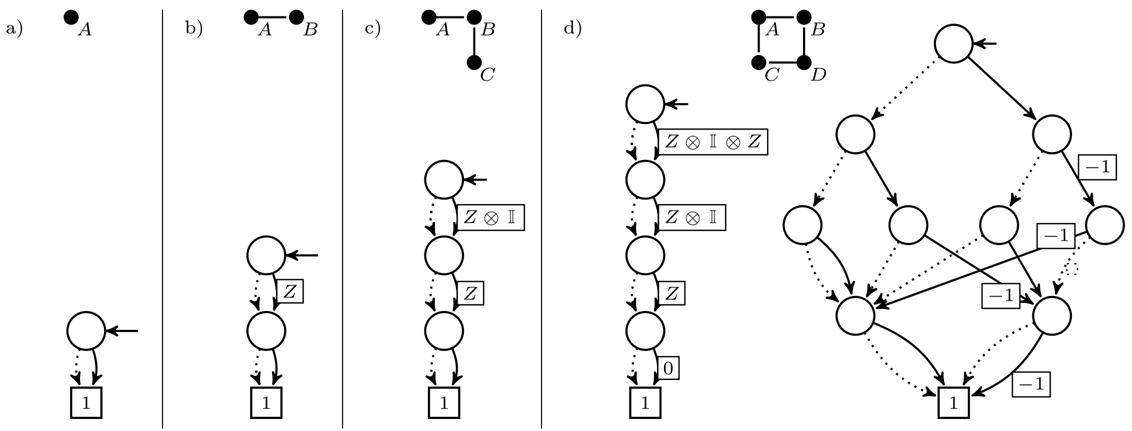


Figure B.1: Construction of the $\langle Z \rangle$-Tower LIMDD for the 4-qubit cluster state, by iterating over the vertices in the graph, as described in the proof of Theorem B.1. (a) First, we consider the single-qubit graph state, which corresponds to a the subgraph containing only vertex A. (b) 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 $Z_B \otimes \mathbb{I}[A]$, since vertex C is connected to vertex B (yielding the Z operator) but not to A (yielding the identity operator $\mathbb{I}$). (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.

$X^{a_n} \otimes \dots \otimes X^{a_1}$ on the root edge, we obtain the coset state $|C + a\rangle$. 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 1/0 and 1/1, 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 $\mathcal{X}_n$. 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 $0a, 0b \in S \implies 0a \oplus 0b \in S$ implies $a, b \in S_0 \implies a \oplus b \in S_0$ and thus S_0 is an length-n bit string vector space. Thus by assumption, we have a procedure to convert it to a Tower-LIMDD in $\mathcal{X}_n$. Convert it into a Tower-LIMDD in $\mathcal{X}_{n+1}$ 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 the root 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 $P = P_n \otimes \dots \otimes P_1$ where $P_j = X$ if $u_j = 1$ and $P_j = \mathbb{I}$ if $u_j = 0$.

We now show $\mathcal{V}_n \subseteq \mathcal{X}_n$, 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 $V = \{0\} \subseteq \{0,1\}^1$.

(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 $|v\rangle = |0\rangle |v_0\rangle$ 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 $|v\rangle = |0\rangle |v_0\rangle + |1\rangle \otimes P |v_0\rangle$. By the observations above, this is a coset state, corresponding to the vector space $V = \{0x | x \in V_0\} \cup \{1(ux) | x \in V_0\}$ 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 $|v_0\rangle$.

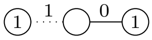
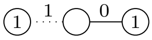
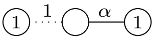
□

Lastly, we prove the stabilizer-state case, showing that they are exactly equivalent to the $\langle$PAULI$\rangle$-Tower-LIMDD, as defined in Definition 3.3. For this, we first need Lemma B.1 and Lemma B.2, which state that, if one applies a Clifford gate to a $\langle$PAULI$\rangle$-Tower-LIMDD, the resulting state is another $\langle$PAULI$\rangle$-Tower-LIMDD. First, Lemma B.1 treats the special case of applying a gate to the top qubit; then Lemma B.2 treats the general case of applying a gate to an arbitrary qubit.

Lemma B.1. Let $|\varphi\rangle$ be an n-qubit stabilizer state which is represented by a $\langle$PAULI$\rangle$-Tower-LIMDD as defined in Definition 3.3. 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 $U|\varphi\rangle$ is still represented by a $\langle$PAULI$\rangle$-Tower-LIMDD.

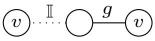
Proof. The proof is on the number n of qubits.

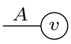
Base case: $n = 1$. For $n = 1$, there are six single-qubit stabilizer states $|0\rangle, |1\rangle$ and $(|0\rangle + \alpha|1\rangle)/\sqrt{2}$ for $\alpha \in \{\pm 1, \pm i\}$. There are precisely represented by Pauli-Tower-LIMDDs with high edge label factor $\in \{0, \pm 1, \pm i\}$ as follows:

- for $|0\rangle$: 
- for $|1\rangle$: $A \cdot$  where $A \propto X$ or $A \propto Y$
- for $(|0\rangle + \alpha\,|1\rangle)/\sqrt{2})$: 

Since the H and S gate permute these six stabilizer states, $U\,|\varphi\rangle$ is represented by a $\langle$Pauli$\rangle$-Tower-LIMDD if $|\varphi\rangle$ is.

Induction case. For $n > 1$, we first consider $U = S$ and $U = \text{CNOT}$. 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 $A \mapsto X_k A$. Next, the root edge label is updated to $URU^\dagger$, 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 $\in \{0, \pm 1, \pm i\}$, we conclude that both these gates yield a $\langle$Pauli$\rangle$-Tower-LIMDD. Finally, for the Hadamard, we decompose $|\varphi\rangle = |0\rangle \otimes |\psi\rangle + \alpha\,|1\rangle \otimes P\,|\psi\rangle$ for some $(n-1)$-qubit stabilizer state $|\psi\rangle$, $\alpha \in \{0, \pm 1, \pm i\}$ and P is an $(n-1)$-qubit Pauli string. Now we note that $H\,|\varphi\rangle \propto |0\rangle \otimes |\psi_0\rangle + |1\rangle \otimes |\psi_1\rangle$ where $|\psi_x\rangle := (\mathbb{I} + (-1)^x \alpha P)\,|\psi\rangle$ with $x \in \{0, 1\}$. Now we consider two cases, depending on whether P commutes with all stabilizers of $|\psi\rangle$:

(a) There exist a stabilizer g of $|\psi\rangle$ which anticommutes with P. We note two things. First, $\langle\psi|P|\psi\rangle = \langle\psi|Pg|\psi\rangle = \langle\psi|g \cdot (-P)|\psi\rangle = -\langle\psi|P|\psi\rangle$, hence $\langle\psi|P|\psi\rangle = 0$. It follows from Lemma 15 of [125] that $|\psi_x\rangle$ is a stabilizer state, so by the induction hypothesis it can be written as a $\langle$Pauli$\rangle$-Tower-LIMDD. Let v be the root node of this LIMDD. Next, we note that $g\,|\psi_0\rangle = g(\mathbb{I} + \alpha P)\,|\psi\rangle = (\mathbb{I} - \alpha P)g\,|\psi\rangle = |\psi_1\rangle$. Hence,  is the root node of a $\langle$Pauli$\rangle$-Tower-LIMDD for $H\,|\varphi\rangle$.

(b) All stabilizers of $|\psi\rangle$ commute with P. Then $(-1)^y P$ is a stabilizer of $|\psi\rangle$ for either $y = 0$ or $y = 1$. Hence, $|\psi_x\rangle = (\mathbb{I} + (-1)^x \alpha P)\,|\psi\rangle = (1 + (-1)^{x+y}\alpha)\,|\psi\rangle$. Therefore, $|\varphi\rangle = |a\rangle \otimes |\psi\rangle$ where $|a\rangle := (1 + (-1)^y \alpha)\,|0\rangle + (1 + (-1)^{y+1}\alpha\,|1\rangle)$. It is not hard to see that $|a\rangle$ is a stabilizer state for all choices of $\alpha \in \{0, \pm 1, \pm i\}$. By the induction hypothesis, both $|a\rangle$ and $|\psi\rangle$ can be represented as $\langle$Pauli$\rangle$-Tower-LIMDDs. We construct a $\langle$Pauli$\rangle$-Tower-LIMDD for $H\,|\varphi\rangle$ by replacing the leaf of the LIMDD of $|a\rangle$ by the root node of the LIMDD of $|\psi\rangle$, and propagating the root edge label of $|\psi\rangle$ upwards. Specifically, if the root edge of $|a\rangle$ is  with $v = \textcircled{1} \overset{1}{\cdots} \bigcirc \overset{\beta}{\rule[0.5ex]{2em}{0.4pt}} \textcircled{1}$, and if the root edge of $|\psi\rangle$ is $\overset{B}{\rule[0.5ex]{2em}{0.4pt}} \textcircled{w}$, then a $\langle$Pauli$\rangle$-Tower-LIMDD for $H|\varphi\rangle$ has root node $\textcircled{$\mathbb{I}$} \overset{w}{\cdots} \bigcirc \overset{\beta\mathbb{I}}{\rule[0.5ex]{3em}{0.4pt}} \textcircled{w}$ and has root edge label $A \otimes B$.

$\square$

Lemma B.2. Let $|\varphi\rangle$ be an n-qubit state state represented by a $\langle$Pauli$\rangle$-Tower-LIMDD, as defined in Definition 3.3. Let U be either a Hadamard gate, an S gate or a CNOT gate. Then $U|\varphi\rangle$ is a state which is also represented by a $\langle$Pauli$\rangle$-Tower-LIMDD.

Proof. The proof is by induction on n. The case $n = 1$ is covered by Lemma B.1. Suppose that the induction hypothesis holds, and let $|\varphi\rangle$ be an $n+1$-qubit state represented by a $\langle$Pauli$\rangle$-Tower-LIMDD. First, we note that a CNOT gate CX_c^t can be written as $CX_c^t = (H \otimes H)CX_t^c(H \otimes H)$, 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 $\langle$Pauli$\rangle$-Tower-LIMDD, according to Lemma B.1.

(b) U does not affect the top qubit. Suppose $|\varphi\rangle = |0\rangle \otimes |\varphi_0\rangle + |1\rangle \otimes \alpha P |\varphi_0\rangle$ (with P a Pauli string and $\alpha \in \{0, \pm 1, \pm i\}$). Then $U|\varphi\rangle = |0\rangle \otimes U|\varphi_0\rangle + |1\rangle \otimes (\alpha U P U^\dagger) U |\varphi_0\rangle$. 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 $\langle$Pauli$\rangle$-Tower-LIMDD. Let $\overset{A}{\rule[0.5ex]{2em}{0.4pt}} \textcircled{v}$ be the root edge of this $\langle$Pauli$\rangle$-Tower-LIMDD, representing $U|\varphi_0\rangle$. Then $U|\varphi\rangle$ is represented by the root edge $\overset{\mathbb{I} \otimes A}{\rule[0.5ex]{4em}{0.4pt}} \textcircled{w}$, where w is the node $\textcircled{v} \overset{\mathbb{I}}{\cdots} \bigcirc \overset{\alpha A^{-1} U P U^\dagger A}{\rule[0.5ex]{8em}{0.4pt}} \textcircled{v}$. The label $\alpha A^{-1} U P U^\dagger A$ is a Pauli LIM, and may therefore be used as the label on the high edge of w.

$\square$

Finally, we show that stabilizer states are precisely the $\langle$Pauli$\rangle$-Tower-LIMDDs.

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

Proof. We first prove that each stabilizer state is represented by a $\langle\textsc{Pauli}\rangle$-Tower-LIMDD. We recall that each stabilizer state can be obtained as the output state of a Clifford circuit on input state $|0\rangle^{\otimes n}$. Each Clifford circuit can be decomposed into solely the gates H, S and CNOT. The state $|0\rangle^{\otimes n}$ is represented by a $\langle\textsc{Pauli}\rangle$-Tower-LIMDD. According to Lemma B.2, applying an H, S or CNOT gate to a $\langle\textsc{Pauli}\rangle$-Tower-LIMDD results a state represented by another $\langle\textsc{Pauli}\rangle$-Tower-LIMDD. One can therefore apply the gates of a Clifford circuit to the initial state $|0\rangle$, and obtain a $\langle\textsc{Pauli}\rangle$-Tower-LIMDD for every intermediate state, including the output state. Therefore, every stabilizer state is represented by a $\langle\textsc{Pauli}\rangle$-Tower-LIMDD.

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

Appendix C

Advanced LIMDD algorithms

C.1 Measuring an arbitrary qubit

Algorithm 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 Algorithm 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 Algorithm 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 $\Pi_k^y = \mathbb{I}[n-k] \otimes |y\rangle\langle y| \otimes \mathbb{I}[k-1]$ 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. Algorithm 19 realizes this. It does so by traversing the LIMDD until a node v with $\mathsf{idx}(v) = k$ is reached. It then returns an edge to a new node by calling MakeEdge(follow$(0, e), 0)$ to project onto $|0\rangle$ or MakeEdge$(0,$ follow$(1, e))$ to project onto $|1\rangle$, 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., $P_k \in \{\mathbb{I}[2], Z\}$). Otherwise, if P_k is

Algorithm 18 Compute the probability of observing $|y\rangle$ when measuring the k-th qubit of the state $|e\rangle$. 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 $\textsc{Measure}(e, 0, n)$. The procedure $\textsc{SquaredNorm}(e, y, k)$ computes the scalar $\langle e| (\mathbb{I} \otimes |y\rangle \langle y| \otimes \mathbb{I}) |e\rangle$, i.e., computes the squared norm of the state $|e\rangle$ after the k-th qubit is projected to $|y\rangle$. For readability, we omit calls to the cache, which implement dynamic programming.

```
1: procedure $\textsc{MeasurementProbability}(\textsc{Edge}\ e \xrightarrow{\lambda P_n \otimes P'} \textcircled{$v$},\ y \in \{0,1\},\ k \in [1...\mathsf{idx}(v)])$
2:     if $n = k$ then
3:         $p_0 := \textsc{SquaredNorm}(\textsc{follow}(0, e))$
4:         $p_1 := \textsc{SquaredNorm}(\textsc{follow}(1, e))$
5:         return $p_j/(p_0 + p_1)$ where $j = 0$ if $P_n \in \{\mathbb{I}, Z\}$ and $j = 1$ if $P_n \in \{X, Y\}$
6:     else
7:         $p_0 := \textsc{SquaredNormProjected}(\textsc{follow}(0, e), y, k)$
8:         $p_1 := \textsc{SquaredNormProjected}(\textsc{follow}(1, e), y, k)$
9:         return $(p_0 + p_1)/\textsc{SquaredNorm}(e)$
10: procedure $\textsc{SquaredNorm}(\textsc{Edge} \xrightarrow{\lambda P} \textcircled{$v$})$
11:     if $n = 0$ then return $|\lambda|^2$
12:     $s := \textsc{Add}(\textsc{SquaredNorm}(\textsc{follow}(0, \xrightarrow{\mathbb{I}} \textcircled{$v$})), \textsc{SquaredNorm}(\textsc{follow}(1, \xrightarrow{\mathbb{I}} \textcircled{$v$})))$
13:     return $|\lambda|^2 s$
14: procedure $\textsc{SquaredNormProjected}(\textsc{Edge}\ e \xrightarrow{\lambda P_n \otimes P'} \textcircled{$v$},\ y \in \{0,1\},\ k \in [1...\mathsf{idx}(v)])$
15:     $b := (P_n \in \{X, Y\})$                    ▷ i.e., $b = 1$ iff $P_n$ is Anti-diagonal
16:     if $n = 0$ then
17:         return $|\lambda|^2$
18:     else if $n = k$ then
19:         return $\textsc{SquaredNorm}(\textsc{follow}(b \oplus y, e))$
20:     else
21:         $\alpha_0 := \textsc{SquaredNormProjected}(\textsc{follow}(0, \xrightarrow{\mathbb{I}} \textcircled{$v$}), b \oplus y, k)$
22:         $\alpha_1 := \textsc{SquaredNormProjected}(\textsc{follow}(1, \xrightarrow{\mathbb{I}} \textcircled{$v$}), b \oplus y, k)$
23:         return $|\lambda|^2 \cdot (\alpha_0 + \alpha_1)$
```

an antidiagonal operator (i.e, $P_k \in \{X, Y\}$), have $\Pi_k^b \cdot P = P\Pi_k^{(1-b)}$. The algorithm applies this correction on Line 2. The resulting state should still be normalized as shown in Sec. 3.3.3.1.

Sampling. To sample from a quantum state in the computational basis, simply repeat the three-step measurement procedure outlined in Sec. 3.3.3.1 n times: once for each

Algorithm 19 Project the state given by LIMDD $\xrightarrow{A}\textcircled{v}$ to state $|b\rangle$ for qubit k, i.e., produce a LIMDD representing the state $(\mathbb{I}[n-k] \otimes |b\rangle\langle b| \otimes \mathbb{I}[k-1]) \cdot A|v\rangle$, with $A = \lambda P_n \otimes \cdots \otimes P_1$.

```
procedure UpdatePostMeas(Edge $\xrightarrow{\lambda P_n \otimes \cdots \otimes P_1}\textcircled{v}$, $k \in [1...\mathsf{idx}(v)]$, $b \in \{0,1\}$)
    $b' := x \oplus b$ where $x = 0$ if $P_k \in \{\mathbb{I}, Z\}$ and $x = 1$ if $P_k \in \{X, Y\}$   ▷ flip $b$ if $P_k$ is anti-diagonal
    if $(v, k, b') \in$ Cache then return Cache$[v, k, b']$
    $n := \mathsf{idx}(v)$
    if $n = k$ then
        $e :=$ MakeEdge$((1 - b') \cdot \mathsf{low}_v,\ b' \cdot \mathsf{high}_v)$   ▷ Project $|v\rangle$ to $|b'\rangle\langle b'| \otimes \mathbb{I}[2]^{\otimes n-1}$
    else   ▷ $n \neq k$:
        $e :=$ MakeEdge(UpdatePostMeas$(\mathsf{low}_v, k, b')$, UpdatePostMeas$(\mathsf{high}_v, k, b')$)

    Cache$[v, k, b'] := e$
    return $e$
```

qubit.

Strong simulation. To compute the probability of observing a given bit-string $x = x_n \dots x_1$, first compute the probability p_n of observing $|x_n\rangle$; then update the LIMDD to outcome x_n, obtaining a new, smaller LIMDD. On this new LIMDD, compute the probability p_{n-1} of observing $|x_{n-1}\rangle$, and so forth. Note that, because the LIMDD was updated after observing the measurement outcome $|x_n\rangle$, p_{n-1} is the probability of observing x_{n-1} given that the top qubit is measured to be x_n. Then the probability of observing the string x is the product $p = p_1 \cdots p_n$.

C.2 Applying Hadamards to stabilizer states in polynomial time

We show that, using the algorithms that we have given,[*] a Hadamard can be applied to a stabilizer state in polynomial time (Theorem C.1). Together with the algorithms for the other Clifford gates, presented in Sec. 3.3.3.2, this shows that all Clifford gates can be applied to stabilizer states in polynomial time. We emphasize that our algorithms do not invoke existing algorithms that are tailored to applying a Hadamard

---

[*] We make minor modifications to the Add algorithm, which are presented in Theorem C.1

to a stabilizer state; instead, the LIMDD algorithms are inherently polynomial-time for this use case. The key ingredient is Lemma C.3, which describes situations in which the ADD procedure adds two stabilizer states in polynomial time. It shows that only $5n$ distinct calls to ADD are made. Our algorithms are polynomial-time because of the dynamic programming effected by the caching of previously computed results, as described in Sec. 3.3.3.3, which, in this case, makes sure only $5n$ recursive calls are made.

Theorem C.1. The algorithm HGATE($\xrightarrow{A}\textcircled{v}, k$) (Algorithm 10) takes polynomial time when the input edge $\xrightarrow{A}\textcircled{v}$ represents a stabilizer state.

Proof. Due to the cache, the algorithm HGATE effects only one recursive call per node. The LIMDD of a stabilizer state has one node on each of the n layers, so there are at most n recursive calls.

When the algorithm arrives at layer k, it makes two calls to ADD. Both calls are of the form ADD($\xrightarrow{\lambda P}\textcircled{v'}, \xrightarrow{\omega Q}\textcircled{v'}$) where $\lambda, \omega \in \{0, \pm 1, \pm i\}$, where P and Q are Pauli strings, and v' is a node representing a stabilizer state, namely v' is the node at the $(k-1)$-th level of the LIMDD. This satisfies the conditions of Lemma C.3; therefore, both calls to ADD make at most $5k = \mathcal{O}(n)$ recursive calls in total. Each recursive call to ADD may invoke the MAKEEDGE procedure, which runs in time $\mathcal{O}(n^3)$, yielding a total worst-case runtime of $\mathcal{O}(n^4)$. Since there are two calls to ADD, the total runtime of HGATE is also $\mathcal{O}(n^4)$.

Lastly, for completeness we note that the call to ADD($\xrightarrow{\lambda P}\textcircled{v'}, \xrightarrow{\omega Q}\textcircled{v'}$) may have $\lambda = 0$ or $\omega = 0$, i.e., one of the operands may be the zero vector. For readability, we have presented the ADD algorithm (Algorithm 9) without treatment of this case, when one of the edges is the zero vector. For the purposes of this proof, we therefore add the following two lines to the ADD algorithm:

```
procedure ADD(EDGE $e = \xrightarrow{\alpha P}\textcircled{v}$, EDGE $f = \xrightarrow{\beta Q}\textcircled{w}$)
    ...
    if $\alpha = 0$ then return $\xrightarrow{\beta Q}\textcircled{w}$
    if $\beta = 0$ then return $\xrightarrow{\alpha P}\textcircled{v}$
    ...
```

These simple checks are also present in the C++ implementation presented in Chapter 4 and is routine in DD implementations, such as the matrix addition algorithm described by Miller and Thornton [227]. Consequently, a call to Add($\xrightarrow{\alpha P}\textcircled{v}$, $\xrightarrow{\beta Q}\textcircled{w}$) runs in $\mathcal{O}(1)$ time if $\alpha = 0$ or $\beta = 0$. □

We now prepare Lemma C.3, which is the main technical ingredient. It states that all the recursive calls to Add effect only five different cache entries at any given level of the LIMDD. To this end, the strategy is (1) to look closely at which recursive calls made by Add; (2) to look closely at when a cache hit is achieved; and (3) to inspect the Follow procedure.

The recursive calls of Add. First, we will find a good description of the set of recursive calls made by a call to Add. We note that each call to Add makes two recursive calls. Specifically, when it is called with parameters Add($\xrightarrow{\alpha P}\textcircled{v}$, $\xrightarrow{\beta Q}\textcircled{v}$), it makes two recursive calls, of the following form,

$$\textsc{Add}(\textsc{follow}(x, \xrightarrow{\alpha P}\textcircled{v}), \textsc{follow}(x, \xrightarrow{\beta Q}\textcircled{v})) \qquad \text{for } x \in \{0,1\} \tag{C.1}$$

These calls subsequently call Add again, recursively. Let us temporarily forget that some of these calls may not happen because a cache hit preempts them (namely, the Add does not recurse in the cache of a cache hit). Then the set of recursive calls to Add is described by calls of the following form,

$$\textsc{Add}(\textsc{follow}(x, \xrightarrow{\alpha P}\textcircled{v}), \textsc{follow}(x, \xrightarrow{\beta Q}\textcircled{v})) \quad \text{for } x \in \{0,1\}^\ell \text{ for } 0 \le \ell \le n \tag{C.2}$$

Cache hits of Add. Inspecting the algorithm Add (Algorithm 9) in Sec. 3.3.3.3, we see that a call to Add with parameters ($\xrightarrow{A}\textcircled{v}$, $\xrightarrow{B}\textcircled{v}$) effects a cache hit if and only if Add was previously called with ($\xrightarrow{C}\textcircled{v}$, $\xrightarrow{D}\textcircled{v}$) satisfying $A^{-1}B\left|v\right\rangle = C^{-1}D\left|v\right\rangle$. Therefore, let us associate a given call to Add($\xrightarrow{\alpha P}\textcircled{v}$, $\xrightarrow{\beta Q}\textcircled{v}$) with the operator $\alpha^{-1}\beta P^{-1}Q$. Then a call to Add with associated operator U will effect a cache hit if a previous call to Add was associated with the same operator U.[†]

The Follow procedure. We now turn to the Follow procedure. The procedure follow$(x, \xrightarrow{\alpha P}\textcircled{v})$ outputs an edge $\xrightarrow{A}\textcircled{t}$, labeled with some label A. Let $L(x, \xrightarrow{\alpha P}\textcircled{v})$ be the function which outputs this label, i.e., $L(x, \xrightarrow{\alpha P}\textcircled{v}) = A$. In this paragraph, we aim to find a closed-form expression for $L(x, \xrightarrow{\alpha P}\textcircled{v})$ in the case

---

[†] More precisely, a call to Add associated with U effects a cache hit if and only if a previous call was associated with an operator U' satisfing $U \cdot \mathrm{Stab}(\varphi) = U' \cdot \mathrm{Stab}(\varphi)$. Here $U \cdot \mathrm{Stab}(\varphi)$ is the coset obtained by left-multiplying the group $\mathrm{Stab}(\varphi)$ with U. Therefore, the condition $U = U'$, named above, is sufficient, but not necessary.

of Tower Pauli-LIMDDs. If the node v is clear from context, we will write simply $L(x, P)$.

It is useful to conceive of the Follow procedure as traversing a path from v to t of length $\ell = |x|$. Then the label $L(x, \xrightarrow{\alpha P}\textcircled{v})$ is the product of the LIMs on the edges that were traversed (including the label P), after which we discard the most significant ℓ qubits. More precisely, for any Pauli string $A = \alpha P_n \otimes \cdots \otimes P_1$, denote with $A^{(\ell)} = \alpha P_{n-\ell} \otimes \cdots \otimes P_1$ the least significant $(n - \ell)$ gates of A, so that, e.g., $A = P_n \otimes P_{n-1} \otimes P^{(2)}$. (In other words, $A^{(\ell)}$ *discards* the ℓ most significant qubits of A). Then, if the Follow procedure traverses edges $e_1, e_2, \ldots, e_\ell$, labeled with LIMs $A_1, A_2, \ldots, A_\ell$, respectively, then

$$L(x, \xrightarrow{A_1}\textcircled{v}) = \lambda A_1^{(\ell)} \cdot A_2^{(\ell)} \cdots A_\ell^{(\ell)} \qquad \text{for some } \lambda \in \mathbb{C} \tag{C.3}$$

Here the factor λ depends only on x and on the operators of A_1 that were discarded; we give a closed formula for λ below. For example, if $x = 1$ and $P_n = Z$, then $\lambda = -1$. In summary, $L(x, \xrightarrow{A}\textcircled{v})$ is the product of (1) the labels on the traversed edges and (2) a phase λ.

Moreover, the ℓ most significant operators of P influence which path is traversed in the following way. For a pauli string P, let $\chi(P) = \chi_1(P) \ldots \chi_\ell(P) \in \{0, 1\}^\ell$ be the string defined by $\chi_j(P) = 0$ if $P_{n-j+1} \in \{I, Z\}$ and $\chi_j(P) = 1$ otherwise, i.e., if $P_{n-j+1} \in \{X, Y\}$. To be clear, $P^{(\ell)}$ isolates the $n - \ell$ *least significant* qubits, whereas $\chi(P)$ depends on the ℓ *most significant* qubits:

$$P = \underbrace{P_n \otimes \cdots \otimes P_{n-\ell+1}}_{\chi(P) \text{ depends on this part}} \otimes \overbrace{P_{n-\ell} \otimes \cdots \otimes P_1}^{P^{(\ell)} \text{ yields this part}} \tag{C.4}$$

Then we have

$$L(x, P) = \lambda L(x \oplus \chi(P), \mathbb{I}^{\otimes \ell} \otimes P^{(\ell)}) \tag{C.5}$$

where $\lambda = \langle x| P_n \otimes \cdots \otimes P_{n-\ell+1} |x \oplus \chi(P)\rangle$. Therefore,

$$\textsc{follow}(x, \xrightarrow{P}\textcircled{v}) = \xrightarrow{\lambda L(x \oplus \chi(P), \mathbb{I}^{\otimes \ell} \otimes P^{(\ell)})}\textcircled{t} \quad \text{for some } \lambda \in \{0, \pm 1, \pm i\} \tag{C.6}$$

where t is the destination of the path traversed by $\textsc{follow}(x, \xrightarrow{P}\textcircled{v})$. Lastly, we note that

$$L(x, \mathbb{I}^{\otimes \ell} \otimes P^{(\ell)}) = P^{(\ell)} \cdot L(x, \mathbb{I}^{\otimes n}) \tag{C.7}$$

We have thus reduced the problem of finding a closed-form expression for $L(x, \xrightarrow{\alpha P}\textcircled{v})$ to the problem of obtaining a closed-form expression for $L(x, \mathbb{I})$, to which we now turn. In the following, we let $v_0, \dots, v_n$ be the nodes in the Tower Pauli-limdd, with v_0 the top node and v_n the Leaf node (we say that node v_ℓ is on layer ℓ). For a bit $a \in \{0, 1\}$ and Pauli string P, we use the notation $P^a = P$ if $a = 1$ and $P^a = \mathbb{I}$ if $a = 0$. To avoid multiple superscripts, we write $P^{a,(\ell)} = (P^a)^{(\ell)}$ for a bit a and an integer ℓ.

Lemma C.1. Let v be the root node of an n-qubit Tower LIMDD and denote with A_j the label of the (unique) high edge from layer $j - 1$ to layer j in this Tower LIMDD. Let $x \in \{0, 1\}^\ell$. Then there are predicates $V_1, \dots, V_\ell$ such that (1) for each $1 \le j \le \ell$, the predicate $V_j(x)$ can be expressed as the XOR of (a subset of) the variables $x_1, \dots, x_j$; and (2) it holds that

$$L(x, \xrightarrow{\mathbb{I}}\textcircled{v}) = A_1^{V_1(x),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)} \tag{C.8}$$

Proof. We observed above (in Equation C.3) that $L(x, \mathbb{I})$ is the product of the $P^{(\ell)}$ for each label P encountered on the edges traversed by FOLLOW$(x, \mathbb{I})$. For a layer $1 \le j \le \ell$, let $V_j(x)$ be the predicate which is true iff the high edge from layer $j - 1$ to j is traversed by FOLLOW$(x, \xrightarrow{\mathbb{I}}\textcircled{v})$. Recall that the low edges of a LIMDD are labeled with the identity operator $\mathbb{I}$. It follows that, in a Tower-LIMDD, $L(x, \mathbb{I}) = A_1^{V_1(x),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)}$, thus settling claim (1).

We now show that $V_j(x)$ can be expressed as the XOR of (a subset of) the variables $x_1, \dots, x_j$, which proves the lemma. The proof is by induction on the layer index. The induction hypothesis in step j is that the predicates $V_1(x), \dots, V_j(x)$ can each be written as a XOR over the variables $x_1, \dots, x_j$.

Base case. For the base case, we observe that $V_1(x) = x_1$; namely, if $x_1 = 1$, then from layer 0 to layer 1, the path traverses the high edge; otherwise the low edge.

Induction step. Assume the induction hypothesis and consider V_{j+1}. We claim that

$$V_{j+1} = x_{j+1} \oplus (\chi_{j+1}(A_1) \wedge V_1) \oplus \cdots \oplus (\chi_{j+1}(A_j) \wedge V_j) \tag{C.9}$$

Namely, for each visited high edge with label A, the bit $\chi_{j+1}(A)$ “flips” the instruction for the path to traverse the high or low edge at layer $j+1$. Lastly, since the bits $\chi_{j+1}(A) \in \{0,1\}$ are constants defined by the LIMDD, and the expressions $V_1, \dots, V_j$ are XORs over the variables $x_1, \dots, x_j$, it follows that V_{j+1} is a XOR over the variables $x_1, \dots, x_{j+1}$. ☐

Lemma C.2. Let v be the root node of an n-qubit Tower PAULI-LIMDD. Let $x, y \in \{0,1\}^\ell$ for some $0 \le \ell \le n$. Then $L(x, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) \cdot L(y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v})^{-1} = \pm L(x \oplus y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v})$.

Proof. Let $V_1, \dots, V_\ell$ be the predicates determining L as in Equation C.8. Then,

$$
\begin{aligned}
L(x, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) &= A_1^{V_1(x),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)} && \text{(C.10)}\\
L(y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) &= A_1^{V_1(y),(\ell)} \cdots A_\ell^{V_\ell(y),(\ell)} && \text{(C.11)}\\
L(x \oplus y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) &= A_1^{V_1(x\oplus y),(\ell)} \cdots A_\ell^{V_\ell(x\oplus y),(\ell)} && \text{(C.12)}\\
&= A_1^{V_1(x)\oplus V_1(y),(\ell)} \cdots A_\ell^{V_\ell(x)\oplus V_\ell(x),(\ell)} && \text{(C.13)}\\
&= \pm\, A_1^{V(x),(\ell)} A_1^{V_1(y),(\ell)} \cdots A_\ell^{V_\ell(y),(\ell)} A_\ell^{V_\ell(y),(\ell)} && \text{(C.14)}
\end{aligned}
$$

Here, in Equation C.13, we have used the fact that, since $V_j(x \oplus y)$ is simply a XOR over some of its inputs, we have

$$
V_j(x \oplus y) = V_j(x) \oplus V_j(y) \tag{C.15}
$$

In Equation C.14, we have used the fact that $A^{a \oplus b} = \pm A^a \cdot A^b$. Namely, we have $A = \lambda P$ for some $\lambda \in \{0, \pm 1, \pm i\}$; thus, if $a = b = 1$ and $\lambda = \pm i$ then $A^2 = -\mathbb{I}$ so $A^{a\oplus b} = \mathbb{I} = -A^a \cdot A^b$; otherwise, if $\lambda \in \{0, \pm 1\}$ or if $a = 0$ or $b = 0$ we have $A^{a \oplus b} = A^a \cdot A^b$. We now obtain $L(x, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) \cdot L(x \oplus y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) = L(y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v})$ by simple algebraic manipulation:

$$
\begin{aligned}
& L(x, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) \cdot L(x \oplus y, \overset{\mathbb{I}}{\text{—}}\!\textcircled{v}) && \text{(C.16)}\\
&= \pm\, \underbrace{A_1^{V_1(x),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)}}_{L(x,\mathbb{I})} \cdot \underbrace{A_1^{V_1(x),(\ell)} \cdot A_1^{V_1(y),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)} \cdot A_\ell^{V_\ell(y),(\ell)}}_{L(x\oplus y)} && \text{(C.17)}\\
&= \pm\, A_1^{V_1(x),(\ell)} \cdot A_1^{V_1(x),(\ell)} \cdot A_1^{V_1(y),(\ell)} \cdots A_\ell^{V_\ell(x),(\ell)} \cdot A_\ell^{V_\ell(x),(\ell)} \cdot A_\ell^{V_\ell(y),(\ell)} && \text{(C.18)}\\
&= \pm\, A_1^{V_1(x)\oplus V_1(x)\oplus V_1(y),(\ell)} \cdots A_\ell^{V_\ell(x)\oplus V_\ell(x)\oplus V_\ell(y),(\ell)} && \text{(C.19)}\\
&= \pm\, A_1^{V_1(y),(\ell)} \cdots A_\ell^{V_\ell(y),(\ell)} = \pm L(y, \mathbb{I}) && \text{(C.20)}
\end{aligned}
$$

We obtain Equation C.18 by grouping like terms; this “shuffling” is possible because

Pauli operators either commute or anticommute. The statement $L(x,\mathbb{I})\cdot L(y,\mathbb{I})^{-1} = \pm L(x\oplus y,\mathbb{I})$ follows from Equation C.20. $\square$

Lemma C.3. Let $v=v_0$ be a node in a Tower Pauli-LIMDD representing a stabilizer state and let P,Q Pauli strings. Then a call to ADD($\xrightarrow{\alpha P}\textcircled{v}$, $\xrightarrow{\beta Q}\textcircled{v}$) invokes only at most $5n$ recursive calls to ADD.

Proof. We observed in Equation C.2 that when ADD is called with parameters ADD($\xrightarrow{\alpha P}\textcircled{v_0}$, $\xrightarrow{\beta Q}\textcircled{v_0}$), the parameters to the recursive calls are all of the form

$$\text{ADD}(\text{FOLLOW}(x, \xrightarrow{\alpha P}\textcircled{v_0}), \text{FOLLOW}(x, \xrightarrow{\beta Q}\textcircled{v_0})) \quad \text{for some } x\in\{0,1\}^{\ell} \text{ and } 0\leq \ell\leq n \tag{C.21}$$

Using the insights above, we have, for any $x\in\{0,1\}^{\ell}$,

$$\text{FOLLOW}(x, \xrightarrow{\alpha P}\textcircled{v_0}) = \xrightarrow{\alpha\lambda L(x\oplus\chi(P),\mathbb{I}^{\otimes\ell}\otimes P^{(\ell)})}\textcircled{v_ℓ} = \xrightarrow{\alpha\lambda P^{(\ell)}L(x\oplus\chi(P),\mathbb{I})}\textcircled{v_ℓ} \tag{C.22}$$

$$\text{FOLLOW}(x, \xrightarrow{\beta Q}\textcircled{v_0}) = \xrightarrow{\beta\omega L(x\oplus\chi(Q),\mathbb{I}^{\otimes\ell}\otimes R^{(\ell)})}\textcircled{v_ℓ} = \xrightarrow{\beta\omega Q^{(\ell)}L(x\oplus\chi(Q),\mathbb{I})}\textcircled{v_ℓ} \tag{C.23}$$

with $\lambda,\omega\in\{0,\pm1,\pm i\}$. Thus, ADD is called with parameters

$$\text{ADD}\left(\xrightarrow{\alpha\lambda P^{(\ell)}L(x\oplus\chi(P),\mathbb{I})}\textcircled{v_ℓ}, \xrightarrow{\beta\omega R^{(\ell)}L(x\oplus\chi(Q),\mathbb{I})}\textcircled{v_ℓ}\right) \quad \text{for some } \lambda,\omega\in\{0,\pm1,\pm i\} \tag{C.24}$$

Therefore, this call to ADD can be associated with the following operator,

$$\begin{aligned}
&(\alpha\lambda P^{(\ell)}L(x\oplus\chi(P),\mathbb{I}))^{-1}\cdot(\omega Q^{(\ell)}L(x\oplus\chi(Q),\mathbb{I})) && \text{(C.25)}\\
=&\alpha^{-1}\lambda^{-1}\beta\omega P^{(\ell)}Q^{(\ell)}L(x\oplus\chi(P),\mathbb{I})L(x\oplus\chi(Q),\mathbb{I})^{-1} && \text{(C.26)}\\
=&\theta P^{(\ell)}Q^{(\ell)}L(\chi(P)\oplus\chi(Q),\mathbb{I}) && \text{(C.27)}
\end{aligned}$$

for some $\theta\in\mathbb{C}$. In Equation C.27 we have used Lemma C.2 to obtain

$$L(x\oplus\chi(P),\mathbb{I})^{-1}\cdot L(x\oplus\chi(Q),\mathbb{I}) = \pm L(\chi(P)\oplus\chi(Q),\mathbb{I}) \tag{C.28}$$

Recall that in a Tower PAULI-LIMDD, all edge weights are in $\{0,\pm1,\pm i\}$, so in particular we have $\theta\in\{0,\pm1,\pm i\}$. We observe that this operator depends on the level ℓ and only the phase θ depends on x. That is to say, $P^{(\ell)}$, $Q^{(\ell)}$ and $L(\chi(P)\oplus\chi(Q),\mathbb{I})$ are fixed for a given level ℓ. It follows that each recursive call to ADD at some level ℓ

is associated with the same operator, modulo some phase $\theta \in \{0, \pm 1, \pm i\}$. Therefore, the cache only stores at most five distinct recursive calls, and will achieve a cache hit on all other recursive calls, at level ℓ. When a cache hit is achieved, the algorithm does not recurse further, and instead terminates the current call. Since the diagram contains n levels, there are at most $5n$ recursive calls in total. $\square$

Appendix D

LIMDDs prepare the W state efficiently

In this section, we prove Theorem 3.6. To this end, we show that LIMDDs can efficiently simulate a circuit family given by McClung [220], which prepares the $|W\rangle$ state when initialized to the $|0\rangle$ state. We thereby show a separation between LIMDD and the Clifford+T simulator, as explained in Sec. 3.3.4.3. Figure Figure D.1 shows the circuit for the case of 8 qubits.

Theorem 3.6. There exists a circuit family C_n such that $C_n\left|0\right\rangle^{\otimes n} = |W_n\rangle$, 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.

Proof. The proof outline is as follows. First, we establish that the LIMDD of each intermediate state (Lemma D.3), as well as of each gate (Lemma D.4), has polynomial size. Second, we establish that the algorithms presented in Sec. 3.3.3 can apply each gate to the intermediate state in polynomial time (Lemma D.8). 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 D.2). For example, the output state has $|x_k\rangle = \left|0\right\rangle^{k-1}\left|1\right\rangle\left|0\right\rangle^{\otimes n-k}$. 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 D.1), and also allow us to upper bound the time taken by the ApplyGate and Add algorithms (in Lemma D.5 for ApplyGate and Lemma D.6 Add).

The theorem follows from Lemma D.8 and Corollary D.1. □

Figure D.1 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 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 Figure D.1. On input $|0\rangle^{\otimes n}$, the circuit's final state is $|W_n\rangle$. We emphasize that the Controlled-X gates are

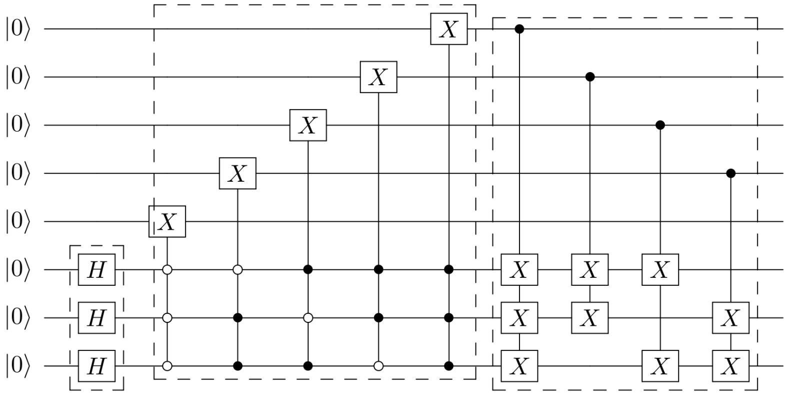


Figure D.1: Reproduced from McClung [220]. A circuit on eight qubits ($n = 8$) which takes as input the $|0\rangle^{\otimes 8}$ state and outputs the $|W_8\rangle$ state. In the general case, it contains $\log n$ Hadamard gates, and its Controlled-X gates act on one target qubit and at most $\log n$ control qubits.

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 D.1. (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 $\alpha\colon \{0,1\}^n \to \mathbb{C}$. Then the *computational basis rank* of $|\psi\rangle$ is $\chi_{\text{comp}}(|\psi\rangle) = |\{x \mid \alpha(x) \neq 0\}|$, the number of nonzero computational basis amplitudes.

Recall that, for a given function $\alpha\colon \{0,1\}^n \to \mathbb{C}$, a string $a \in \{0,1\}^\ell$ induces a *subfunction* $\alpha_y\colon \{0,1\}^{n-\ell} \to \mathbb{C}$, defined as $\alpha_y(x) = \alpha(y,x)$. 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 D.2. (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 $\alpha^\psi\colon \{0,1\}^n \to \mathbb{C}$, as above. Let $\chi_{\text{sub}}(|\psi\rangle, \ell)$ be the number of unique non-zero subfunctions induced by strings of length ℓ, as follows,

$$\chi_{\text{sub}}(|\psi\rangle, \ell) = |\{\alpha_y^\psi\colon \{0,1\}^{n-\ell} \to \mathbb{C} \mid \alpha_y \neq 0, y \in \{0,1\}^\ell\}| \tag{D.1}$$

We define the *subfunction rank* of $|\psi\rangle$ as $\chi_{\text{sub}}(|\psi\rangle) = \max_{\ell=0,\dots n} \chi_{\text{sub}}(|\psi\rangle, \ell)$. 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 $\alpha^U\colon \{0,1\}^{2n} \to \mathbb{C}$.

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 D.3.

Definition D.3 (Prefix of a LIMDD node)**.** For a given string $x \in \{0,1\}^\ell$, consider the path traversed by the FOLLOW$(x, \xrightarrow{R}\textcircled{r})$ subroutine, which starts at the diagram's root edge and ends at a node v on level ℓ. We will say that x is a *prefix* of the node v.

We let 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 pre(v).

Lemma D.1. 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 Figure 3.3). 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 $\ell \geq 0$,

$$R\,|r\rangle = \sum_{x\in\{0,1\}^\ell} |x\rangle \otimes \textsc{follow}(x, \xrightarrow{R}\textcircled{r}) \tag{D.2}$$

Since $\xrightarrow{R}\textcircled{r}$ is the root of our diagram, if x is a prefix of v, then

$$\textsc{follow}(x, \xrightarrow{R}\textcircled{r}) = \text{Labels}(x)\cdot|v\rangle \tag{D.3}$$

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 ℓ (namely, simply the node at which the $\textsc{follow}(x, \xrightarrow{R}\textcircled{r})$ subroutine arrives). Say that the ℓ-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 Equation D.3 and Equation D.2 together, we can express the root node's state in terms of the nodes $v_1, \ldots, v_m$ on level ℓ:

$$R\,|r\rangle = \sum_{k=1}^{m}\sum_{x\in\text{pre}(v_k)} |x\rangle \otimes \textsc{follow}(x, \xrightarrow{R}\textcircled{r}) \tag{D.4}$$

$$= \sum_{k=1}^{m}\sum_{x\in\text{pre}(v_k)} |x\rangle \otimes \text{Labels}(x)\cdot|v_k\rangle \tag{D.5}$$

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 pre(v_k) are pairwise disjoint. Specifically, we show that each node has at least one prefix x such that 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 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., $|v\rangle = \vec{0}$). 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 Definition 3.5), 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 $\mathbb{I}$ 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 Definition 3.5). Namely, if v is a node, then by the low factoring rule, the low edge has label $\mathbb{I}$. Therefore, if this edge points to node v_0, and the high edge is $\xrightarrow{A}\textcircled{v_1}$, then the node v represents $|v\rangle = |0\rangle\,|v_0\rangle + |1\rangle\,A\,|v_1\rangle$ with possibly $A = 0$, so, if $|v_0\rangle \neq \vec{0}$, then $|v\rangle \neq \vec{0}$. 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 $\textsc{follow}(x, \xrightarrow{R}\textcircled{r}) \neq \vec{0}$. We conclude that the equation above contains at least m non-zero contributions. Hence $m \leq \chi_{\text{comp}}(R\,|r\rangle)$, at any level $0 \leq \ell \leq n$. ☐

Lemma D.2. Each intermediate state in the circuit in Figure D.1 (with $n = 2^c$) has $\chi_{\text{comp}}(|\psi\rangle) \leq n$.

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

$$|\psi_{\log n}\rangle = H^{\otimes \log n} \otimes \mathbb{I}^{n-\log n}\,|0\rangle = |+\rangle^{\otimes \log n} \otimes |0\rangle^{\otimes n-\log n} = \frac{1}{\sqrt{n}}\sum_{x=0}^{n-1} |x\rangle\,|0\rangle^{\otimes n-\log n} \tag{D.6}$$

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. ☐

Lemma D.3. The reduced LIMDD of each intermediate state in the circuit in Figure D.1 has polynomial size.

Proof. By Lemma D.1, 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 D.2, the intermediate states in question have polynomial χ_{comp}, so the result follows. ☐

Lemma D.4. The LIMDD of each gate in the circuit in Figure D.1 (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 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 D.5). 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 χ_{sub}, yet the LIMDD simulation is polynomial-time, as shown in Sec. 3.3.3.4.

Lemma D.5. The number of recursive calls made by subroutine ApplyGate$(U, |\psi\rangle)$, is at most $n\chi_{\text{sub}}(U)\chi_{\text{sub}}(|\psi\rangle)$, for any gate U and any state $|\psi\rangle$.

Proof. Inspecting Algorithm 8, we see that every call to ApplyGate$(U, |\psi\rangle)$ produces four new recursive calls, namely ApplyGate(follow(rc, U), follow$(c, |\psi\rangle)$) for $r, c \in \{0, 1\}$. Therefore, the set of parameters in all recursive calls of ApplyGate$(U, |\psi\rangle)$ is precisely the set of tuples (follow(rc, U), follow$(c, |\psi\rangle)$), with $r, c \in \{0, 1\}^\ell$ with $\ell = 0 \dots n$. The terms follow(rc, U) and follow$(c, |\psi\rangle)$ are precisely the subfunctions of U and $|\psi\rangle$, and since there are at most $\chi_{\text{sub}}(U)$ and $\chi_{\text{sub}}(|\psi\rangle)$ of these, the total number of distinct parameters passed to ApplyGate in recursive calls at level ℓ, is at most $\chi_{\text{sub}}(U, \ell) \cdot \chi_{\text{sub}}(|\psi\rangle, \ell) \leq \chi_{\text{sub}}(U) \cdot \chi_{\text{sub}}(|\psi\rangle)$. Summing over the n levels of the diagram, we see that there are at most $n\chi_{\text{sub}}(U)\chi_{\text{sub}}(|\psi\rangle)$ distinct recursive calls in total. As detailed in Sec. 3.3.3.3, the ApplyGate algorithm caches its inputs in such a way that it will achieve a cache hit on a call 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 $\chi_{\text{sub}}(U)$ and $\chi_{\text{sub}}(|\psi\rangle)$ are polynomial, so a polynomial number of recursive calls to 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 ApplyGate. First, Lemma D.6 shows expresses a worst-case upper bound on the number of recursive calls to Add in terms of χ_{sub}. Then Lemma D.7 uses this result to show that, in our circuit, the number of recursive calls is polynomial in n.

Lemma D.6. The number of recursive calls made by the subroutine $\textsc{Add}(|\alpha\rangle, |\beta\rangle)$ is at most $n\chi_{sub}(|\alpha\rangle) \cdot \chi_{sub}(|\beta\rangle)$, if $|\alpha\rangle, |\beta\rangle$ are n-qubit states.

Proof. Inspecting Algorithm 9, every call to $\textsc{Add}(|\alpha\rangle, |\beta\rangle)$ produces two new recursive calls, namely $\textsc{Add}(\textsc{follow}(0, |\alpha\rangle), \textsc{follow}(0, |\beta\rangle))$ and $\textsc{Add}(\textsc{follow}(1, |\alpha\rangle), \textsc{follow}(1, |\beta\rangle))$. It follows that the set of parameters on $n-\ell$ qubits with which Add is called is the set of tuples $(\textsc{follow}(x, |\alpha\rangle), \textsc{follow}(x, |\beta\rangle))$, for $x \in \{0,1\}^\ell$. This corresponds precisely to the set of subfunctions of α and β induced by length-ℓ strings, of which there are $\chi_{\text{sub}}(|\alpha\rangle, \ell)$ and $\chi_{\text{sub}}(|\beta\rangle, \ell)$, respectively. Because the results of previous computations are cached, as explained in Sec. 3.3.3.3, the total number of recursive calls is the number of *distinct* recursive calls. Therefore, we get the upper bound of $\chi_{\text{sub}}(|\alpha\rangle) \cdot \chi_{\text{sub}}(|\beta\rangle)$ for each level of the LIMDD. Since the LIMDD has n levels, the upper bound $n\chi_{\text{sub}}(|\alpha\rangle) \cdot \chi_{\text{sub}}(|\beta\rangle)$ follows. ☐

Lemma D.7. The calls to $\textsc{Add}(|\alpha\rangle, |\beta\rangle)$ that are made by the recursive calls to $\textsc{ApplyGate}(U_t, |\psi_t\rangle)$, satisfy $\chi_{\text{sub}}(|\alpha\rangle), \chi_{\text{sub}}(|\beta\rangle) = \text{poly}(n)$.

Proof. We have established that the recursive calls to ApplyGate are all called with parameters of the form $\textsc{ApplyGate}(\textsc{follow}(r, c, U_t), \textsc{follow}(c, |\psi_t\rangle))$ for some $r, c \in \{0,1\}^\ell$. Inspecting Algorithm 8, we see that, within such a call, each call to $\textsc{Add}(|\alpha\rangle, |\beta\rangle)$ has parameters which are both of the form $|\alpha\rangle, |\beta\rangle = \textsc{ApplyGate}(\textsc{follow}(rx, cy, U_t), \textsc{follow}(cy, |\psi_t\rangle))$ for some $x, y \in \{0,1\}$; therefore, the parameters $|\alpha\rangle, |\beta\rangle$ are of the form $|\alpha\rangle, |\beta\rangle = \textsc{follow}(r, c, U_t) \cdot \textsc{follow}(r, |\psi_t\rangle)$. Here $\textsc{follow}(cy, |\psi_t\rangle)$ 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, $\chi_{\text{comp}}(\textsc{follow}(x, |\psi\rangle)) \leq \chi_{\text{comp}}(|\psi\rangle)$. In particular, we have $\chi_{\text{comp}}(\textsc{follow}(cy, |\psi_t\rangle)) \leq \chi_{\text{comp}}(|\psi_t\rangle) = \mathcal{O}(n)$. The matrix $\textsc{follow}(rx, cy, U_t)$ is a subfunction of a permutation gate, and applying such a matrix to a vector cannot increase its computational basis rank, so we have

$$\chi_{\text{sub}}(|\alpha\rangle) = \chi_{\text{sub}}(\textsc{follow}(rx, cy, U_t) \cdot \textsc{follow}(cy, |\psi_t\rangle)) \tag{D.7}$$

$$\leq \chi_{\text{comp}}(\textsc{follow}(rx, cy, U_t) \cdot \textsc{follow}(cy, |\psi_t\rangle)) \leq \chi_{\text{comp}}(\textsc{follow}(cy, |\psi_t\rangle)) \tag{D.8}$$

$$\leq \chi_{\text{comp}}(|\psi_t\rangle) = \mathcal{O}(n) \tag{D.9}$$

This proves the lemma. ☐

Lemma D.8. Each call to ApplyGate($U_t, |\psi_t\rangle$) runs in polynomial time, for any gate U_t in the circuit in Figure D.1 (with $n = 2^c$).

Proof. If U_t is a Hadamard gate, then LIMDDs can apply this in polynomial time by Theorem 3.5, since $|\psi_t\rangle$ is a stabilizer state. Otherwise, U_t is one of the controlled-X gates. In this case there are a polynomial number of recursive calls to ApplyGate, by Lemma D.5. Each recursive call to ApplyGate makes two calls to Add($|\alpha\rangle, |\beta\rangle$), where both α and β are states with polynomial subfunction rank, by Lemma D.7. By Lemma D.6, these calls to Add all complete in time polynomial in the subfunction rank of its arguments. ☐

Corollary D.1. The circuit in Figure D.1 (with $n = 2^c$) can be simulated by LIMDDs in polynomial time.

Appendix E

Proofs of Section 5.3

In this appendix, we prove Theorem 5.1 as reproduced below. We also reproduce Figure 5.2 in Figure E.1, which additionally includes references to the respective lemmas. Chapter 2 and Section 5.2 contain relevant preliminaries on quantum information and QDDs.

Theorem 5.1. The succinctness results in Figure 5.2 hold.

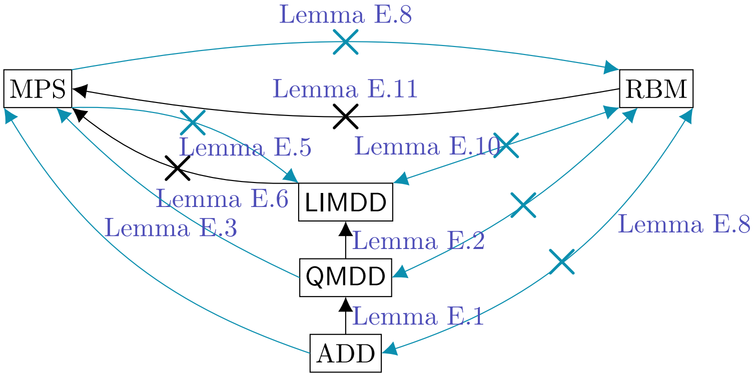


Figure E.1: Succinctness relations between various classical data structures for representing quantum states. Solid arrows $A \rightarrow B$ denote $B \prec_s A$, i.e., B is strictly more succinct than A. Crossed arrows $A \nrightarrow B$ denote a separation $B \not\preceq_s A$; a bidirectional crossed arrow implies incomparability. Blue arrows indicate novel relations that we identified.

Proof. The proofs for individual relations are stated in the lemmas referenced by Figure E.1.

Note that we do not include a proof for every arrow (direction), since several can be derived through transitivity properties. All unlabeled edge (directions) can be derived as follows:

- MPS $\prec_s$ ADD follows from MPS $\prec_s$ QMDD and QMDD $\prec_s$ ADD
- LIMDD $\prec_s$ ADD follows from LIMDD $\prec_s$ QMDD and QMDD $\prec_s$ ADD
- QMDD $\not\preceq_s$ RBM follows from LIMDD $\not\preceq_s$ RBM and LIMDD $\prec_s$ QMDD
- ADD $\not\preceq_s$ RBM follows from LIMDD $\not\preceq_s$ RBM and LIMDD $\prec_s$ ADD
- RBM $\not\preceq_s$ MPS follows from RBM $\not\preceq_s$ ADD and MPS $\prec_s$ ADD
- RBM $\not\preceq_s$ QMDD follows from RBM $\not\preceq_s$ ADD and QMDD $\prec_s$ ADD
- RBM $\not\preceq_s$ LIMDD follows from RBM $\not\preceq_s$ ADD and LIMDD $\prec_s$ ADD

This completes the proof of all stated succinctness relations. □

Lemma E.1. QMDD is exponentially more succinct than ADD.

Proof. Since ADD is a special case of QMDD (Sec. 5.2.2), QMDD is at least as succinct.

Fargier et al. [114] prove an exponential separation in Prop. 10. The proposition itself only mentions a superpolynomial separation; the fact that the separation is in fact exponential is contained in the proof. □

Lemma E.2. LIMDD is exponentially more succinct than QMDD.

Proof. Since QMDD is a special case of LIMDD (Sec. 5.2.2), LIMDD is at least as succinct.

Vinkhuijzen et al. [337] show an exponential separation for so-called 'cluster states.' □

Lemma E.3. MPS is exponentially more succinct than QMDD.

Proof. We show in Section G.4 that MPS is at least as succinct as QMDD, by showing that every QMDD can be translated to MPS in linear time.

We provide a state $|\varphi\rangle$ on n qubits, which has an exponential-sized QMDD, but a polynomial-sized MPS. Let $(x)_2 \in \mathbb{Z}$ be the integer represented by a bit-string $x \in \{0,1\}^n$. The state of interest is

$$|\varphi\rangle = \sum_{x\in\{0,1\}^n} (x)_2 \, |x\rangle = \sum_{x\in\{0,1\}^n} \left(\sum_{j=1}^{n} 2^{j-1} x_j \right) |x\rangle \tag{E.1}$$

Fargier et al. [114] show that this state has exponential-sized QMDD (Prop. 10). On the other hand, it can be efficiently represented by the following MPS of bond dimension 2:

$$A_n^0 = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad A_j^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad A_1^0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{E.2}$$

$$A_n^1 = \begin{bmatrix} 1 & 2^{n-1} \end{bmatrix} \qquad A_j^1 = \begin{bmatrix} 1 & 2^{j-1} \\ 0 & 1 \end{bmatrix} \qquad A_1^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{E.3}$$

Here j ranges from $2 \ldots n-1$. To show this, we can write

$$A_n^{x_n} = \begin{bmatrix} 1 & x_n \cdot 2^{n-1} \end{bmatrix} \qquad A_j^{x_j} = \begin{bmatrix} 1 & x_j \cdot 2^{j-1} \\ 0 & 1 \end{bmatrix} \text{ for } j = 2, ..., n-1 \qquad A_1^{x_1} = \begin{bmatrix} x_1 \\ 1 \end{bmatrix}$$

Hence we can write

$$A_n^{x_n} \cdot \cdots \cdot A_1^{x_1} = \begin{bmatrix} 1 & x_n \cdot 2^{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & \sum_{j=2}^{n-1} x_j \cdot 2^{j-1} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ 1 \end{bmatrix} \tag{E.4}$$

$$= \begin{bmatrix} 1 & x_n \cdot 2^{n-1} \end{bmatrix} \cdot \begin{bmatrix} \sum_{j=1}^{n-1} x_j \cdot 2^{j-1} \\ 1 \end{bmatrix} \tag{E.5}$$

$$= \begin{bmatrix} \sum_{j=1}^{n} 2^{j-1} \cdot x_j \end{bmatrix} \tag{E.6}$$

□

The following quantum state, called $|\mathrm{Sum}\rangle$, will feature in several of the below proofs. Specifically, we will show that RBM and MPS can represent this state efficiently, whereas LIMDDs cannot. A similar state will be used to show that LIMDD does not support the Swap operation. We omit normalization factors, as all data structures are oblivious to them.

$$|\mathrm{Sum}\rangle = |+\rangle^{\otimes n} + \bigotimes_{j=1}^{n} (|0\rangle + e^{i\pi 2^{-j-1}}|1\rangle) \tag{E.7}$$

Lemma E.4. The LIMDD of $|\mathrm{Sum}\rangle$ has size $2^{\Omega(n)}$ for every variable order.

Proof. We compute that the amplitude function for $|\mathrm{Sum}\rangle$ is

$$f(\vec{x}) = 1 + e^{i\pi \sum_{j=1}^{n} x_j \cdot 2^{-j-1}}. \tag{E.8}$$

We note that f is injective and never zero, and indeed that the function $(\vec{x}, \vec{y}) \mapsto \frac{f(\vec{x})}{f(\vec{y})}$ is injective on the domain where $\vec{x} \neq \vec{y}$.

We now study the nodes v at level 1 (with $\mathsf{idx}(v) = 1$) via the subfunctions they represent, considering all variable orders. These nodes represent subfunctions on one variable. So we take out one variable $x_k \in \vec{x} = \{x_1, ..., x_n\}$. Without loss of generality, we may pick x_1 because the summation in Equation E.8 is commutative. For each assignment $\vec{a} \in \{0,1\}^{n-1}$, we obtain the function:

$$f_{\vec{a}}(x_1) = 1 + e^{i\pi \sum_{j=2}^{n} a_j \cdot 2^{-j-1}} \cdot e^{i\pi \cdot 1/4 x_1}.$$

We now show that for any $\vec{a} \neq \vec{c} \in \{0,1\}^{n-1}$ there is no $Q \in \textsc{PauliLIM}_1$ such that $f_{\vec{a}} = Q f_{\vec{c}}$.

Let $Q = \alpha P$ for $\alpha \in \mathbb{C} \setminus \{0\}, P \in \{\mathbb{I}, X, Y, Z\}$, so $f_{\vec{a}} = \alpha P f_{\vec{c}}$. Furthermore, define $\alpha = \alpha(z, x, \vec{a}, x_1) = (-1)^z \cdot \frac{f_{\vec{a}}(x_1 \oplus x = 0)}{f_{\vec{a}}(x_1 \oplus x = 1)}$ for $P = X^x \cdot Z^z$ with $x, z \in \{0,1\}$, absorbing the factor i of Y and -1 of ZX in α. The function α is injective, i.e., $\alpha(s) = \alpha(t)$ implies $s = t$, based on our earlier observations about f.

It follows that each subfunction $f_{\vec{a}}$ requires a separate node at level 1. So there are $\Omega(2^{n-1})$ nodes. ☐

Lemma E.5. There is a family of quantum states with polynomial-size MPS but exponential-size LIMDD.

Proof. MPS require only bond dimension 2 to represent the state $|\mathrm{Sum}\rangle$ as shown by Lemma E.3. However, Lemma E.4 shows that LIMDDs require exponential size to represent the same state. ☐

Lemma E.6. There is a family of quantum states with polynomial-size LIMDD but exponential-size MPS.

Proof. LIMDD can efficiently represent any stabilizer state, but some stabilizer states require exponential-size MPS (in particular, the cluster state, among others [337]. ☐

Lemma E.7. MPS is at least as succinct as QMDD.

Proof. Section G.4 provides a polynomial-time transformation from QMDD to MPS. ☐

For proving the separation between RBM and ADD, we use the seminal Boolean function $IP : \{0,1\}^n \to \{0,1\}, \vec{x} \mapsto \sum_{k=1}^{n/2} x_k x_{k+n/2} \mod 2$ for even n, which computes the inner product between the first half of the input with the second half. Martens et al. [214] show that any RBM requires a number of hidden weights m which is necessarily exponential in m.

Lemma E.8. There is a quantum state that has linear representation both as ADD, and QMDD, and LIMDD, and MPS, but requires exponential space when represented as RBM under any qubit order.

Proof. We will give the proof for the ADD; the result will then follow for QMDD, LIMDD and MPS, since these are at least as succinct as ADD.

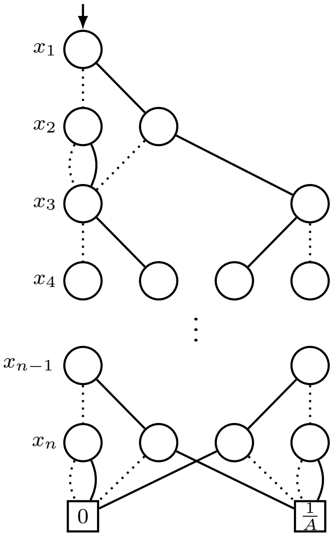


Figure E.2: An ADD for the inner product function IP' from Lemma E.8 made up of stacked blocks, each consisting of a layer of 2 nodes and a layer of 4 nodes. A in the right leaf is the normalization constant from Lemma E.8.

Since we consider the representation size under any qubit variable order, we may as well interleave the order. That is, we consider IP' which equals IP with x_{k+1}

and $x_{k+n/2}$ swapped, i.e. $IP'(x) = x_1x_2 + x_3x_4 + ... + x_{n-1}x_n$. Consider the n-qubit quantum state $|\varphi\rangle$ where $\langle x|\varphi\rangle = IP'(x)/A$ for $x \in \{0, 1\}$, where the normalization factor is $A = \sqrt{\sum_{x\in\{0,1\}} IP'(x)}$. Martens et al. [214] show that any RBM requires a number of hidden weights m which is necessarily exponential in m.

There exists an ADD which represents $|\varphi\rangle$ in $O(n)$ space. This ADD is constructed from stacked blocks of two layers (of 2 and 4 nodes, respectively). The $(k + 1)/2$-th block (counting from 1 from the top) for odd $k = 1, 3, 5, ...$ corresponds to computing the value $x_k \cdot x_{k+1}$ and adding it to the running value of $IP'(x_1, x_2, ..., x_{k-1})$. See Figure E.2. ☐

Lemma E.9. RBM can represent the state $|\text{Sum}\rangle$ with a single hidden node.

Proof. All nodes have bias 0, i.e., $\beta = [0]$ and $\alpha = [0, ..., 0]^T$ (a length-n vector). The weight on the edge between the hidden node and the j-th visible node is $e^{i\pi 2^{-j-1}}$. Then the RBM is defined by the multiplicative term of this hidden node, yielding

$$\psi(\vec{x}) = 1 + e^{w\cdot\vec{x}} = 1 + \prod_{j=1}^{n} e^{x_j i\pi 2^{-j-1}} \tag{E.9}$$

This corresponds exactly with the sum state: $|\psi\rangle = |\text{Sum}\rangle$. ☐

Lemma E.10. There is a state with a RBM of size $\mathcal{O}(n)$ but which requires LIMDD of size $2^{\Theta(n)}$, for every variable order.

Proof. RBM can represent the state $|\text{Sum}\rangle$, by Lemma E.9. However, Lemma E.4 shows that LIMDDs require exponential size to represent this state. ☐

Lemma E.11. There is a family of states with polynomial-size RBM but exponential-size MPS.

Proof. RBM can efficiently represent stabilizer states, as shown by Zhang et al. [365]. Vinkhuijzen et al. [337] show that some stabilizer states require exponential-size MPS (in particular, the cluster state, among others). ☐

Appendix F

Proofs of Section 5.4

In this appendix, we prove Theorem 5.2 and Theorem 5.3 from Section 5.4.

Theorem 5.2 is restated below. The proofs are organized per row of the table, so there is one section for each data structure. Section 5.2 and Chapter 2 contain relevant preliminaries on quantum information and QDDs.

Theorem 5.2. The tractability results in Table 5.2 hold.

We restate the other main result Theorem 5.3 here and provide a proof.

Theorem 5.3. Assuming the exponential time hypothesis, the fidelity of two states represented as LIMDDs or RBMs cannot be computed in polynomial time. The proof uses a reduction from the #EVEN SUBGRAPHS problem [169].

Proof. Lemma F.19 proves that LIMDD does not admit a polynomial time algorithm unless the exponential time hypothesis fails. Corollary F.1 concludes the same for RBM. ☐

F.1 Easy and hard operations for ADD

As noted in Sec. 5.2.2, the decision diagrams are special cases of each other. In particular, ADD specializes QMDD, which specializes LIMDD. From this, it immediately follows that LIMDD $\preceq_s$ QMDD $\preceq_s$ ADD. We also use this fact in the below proofs.

Easy and hard operations for ADD

Lemma F.1. ADD supports **Sample** and **Measure**.

Proof. LIMDD supports these operations (see Lemma F.15). Since ADD specializes LIMDD, it inherits the tractability of these operations ☐

Lemma F.2. ADD supports inner-product $\langle\varphi|\psi\rangle$.

Proof. QMDD supports these operations (see Lemma F.6). Since ADD is a specialization of QMDD, it inherits the tractability of these operations. ☐

Lemma F.3. ADD supports **Addition** and **Equal**.

Proof. See Fargier et al. [113] Table 1 (EQ) and Table 2 (+**BC**). ☐

Lemma F.4. ADD supports **Local**, and hence also **Hadamard**, **X,Y,Z**, T, **Swap** and **CZ**.

Proof. Suppose U is a local gate on k qubits. Then U can be expressed as the sum of 4^k terms, $U = \sum_{x,y\in\{0,1\}^k} a_{xy} |x\rangle\langle y| \otimes \mathbb{I}_{n-k}$. Each of these terms individually can be applied to an ADD in polynomial time ([113] Table 1 **CD**), since they are projections, followed by X gates. Since a constant number of states can be added in polynomial

Table 5.2: Tractability of queries and manipulations on the data structures analyzed in this chapter (single application of the operation). A ✓ means the data structure supports the operation in polytime, a ✓' means supported in randomized polytime, and ✖ means the data structure does not support the operation in polytime. A ○ means the operation is not supported in polytime unless $P = NP$. ? means unknown. The table only considers deterministic algorithms (for some ? a probabilistic algorithm exists, e.g., for **InnerProd** on RBM). Novel results are blue and underlined.

		Queries					Manipulation operations						
		Sample	**Measure**	**Equal**	**InnerProd**	**Fidelity**	**Addition**	**Hadamard**	**X,Y,Z**	**CZ**	**Swap**	**Local**	**T-gate**
Vector		✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ADD	Section F.1	✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
QMDD	Section F.2	✓'	✓	✓	✓	✓	✖	✖	✓	✓	✖	✖	✓
LIMDD	Section F.3	✓'	✓	✓	○	○	✖	✖	✓	✓	✖	✖	✓
MPS	Section F.4	✓'	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
RBM	Section F.5	✓'	?	?	○	○	?	?	✓	✓	✓	?	✓

time in ADDs (Lemma F.3), the result can be computed in polynomial time. Since ADD supports arbitrary k-local gates, in particular it supports all other gates that are mentioned: H, X, Y, Z, T, Swap CZ. □

F.2 Easy and hard operations for QMDD

Lemma F.5. QMDD supports **Equal**, **Sample** and **Measure**.

Proof. LIMDD supports these operations (see Lemma F.15). Since QMDD specializes LIMDD, it inherits the tractability of these operations. □

Lemma F.6. QMDD supports inner product (**InnerProd**) and fidelity (**Fidelity**).

Proof. We show in Section G.4 that a QMDD can be efficiently and exactly translated to an MPS. Since MPS supports inner product and fidelity, the result follows. □

Lemma F.7. QMDD does not support **Addition** in polynomial time.

Proof. Fargier et al. [113] (Thm. 4.9) show that **Addition** is hard for QMDD. □

Lemma F.8. QMDD does not support **Hadamard** in polynomial time and hence neither **Local**.

Proof. By reduction from addition: Take a QMDD root node v with left child a and right child b, then $\textbf{Hadamard}(v) = H\left|v\right\rangle$ is a new node with a left child $\left|a\right\rangle + \left|b\right\rangle$. By choosing $\left|a\right\rangle, \left|b\right\rangle$ to be the states from Fargier et al.'s proof showing that addition is intractable for QMDDs, the state $\left|a\right\rangle + \left|b\right\rangle$ requires an exponential-size QMDD. Since QMDD does not support the Hadamard gate, neither does it support arbitrary local gates (**Local**). □

Lemma F.9. QMDD supports Pauli gates **X,Y,Z** and **T** in polynomial time.

Proof. We will show that we can apply any single-qubit diagonal or anti-diagonal operator $A = \left[\begin{smallmatrix}\alpha & 0\\ 0 & \beta\end{smallmatrix}\right]$ to an QMDD in polynomial time. The result then immediately follows for the special cases of the gates X, Y, Z and T. Applying any diagonal local operator $A = \left[\begin{smallmatrix}\alpha & 0\\ 0 & \beta\end{smallmatrix}\right]$ to the top qubit is easy: simply multiply the weights of the low and high edges of the diagram's root node with respectively α and β. For the anti-diagonal operator A^T, we also swap low with high edges. To apply the local operator on any qubit, simply do the above for all nodes on the corresponding level.

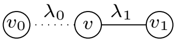
To see that the resulting QMDD indeed represents the state $A \cdot |\psi\rangle$ (or $A^T |\psi\rangle$), consider the amplitude of any basis state $x \in \{0,1\}^n$. The amplitude of $|x\rangle$ in an QMDD is the product of the labels found on the edges while traversing the diagram from root to leaf. In the new diagram, only the weights have changed, whereas the topology has remained the same. If $x_k = 0$ (resp. $x_k = 1$), then, the k-th edge encountered during this traversal is the same in the new diagram as in the old diagram, but the label has been multiplied by α. Otherwise, if $x_k = 1$ (resp. $x_k = 0$), then the label is multiplied by β. All the other weights remain the same. Therefore, the amplitude of x in the new diagram is equal to the old amplitude multiplied by α (resp. β). □

Lemma F.10. QMDD supports controlled-Z in polynomial time.

Proof. Algorithm 20 applies a controlled-Z gate to a QMDD in time linear in the number of nodes in the QMDD. To show that this is the runtime, we consider the number of times the algorithms ApplyControlledZ and ApplyZ are called.

For both these algorithms, say that a call is *trivial* if the result is already in the cache, otherwise a call is *non-trivial*. Then a trivial call completes in constant time (i.e., in time $\mathcal{O}(1)$). Moreover, the number of trivial calls is at most twice the number of non-trivial calls. Therefore, for the purposes of obtaining an asymptotic upper bound on the running time, it suffices to count the number of non-trivial calls to the algorithm.

Thanks to the cache, a given setting of the input parameters (v, a, b) (or (v, t) in the case of ApplyZ) will trigger only one non-trivial call. Therefore, the number of non-trivial calls is equal to the number of distinct input parameters. But here only v varies, so the number of non-trivial calls is at most the number of nodes in the QMDD. This reasoning holds for both algorithms ApplyControlledZ and ApplyZ. Therefore, both subroutines run in time $\mathcal{O}(m)$, for an QMDD which contains m nodes.

The correctness of this algorithm follows from the fact that $CZ_{a,b} |v\rangle = \lambda_0 |v_0\rangle + \lambda_1 Z_b |v_1\rangle$ where node v is represents an $a - qubit$ state  and Z_b means applying the Z gate to the b-th qubit. This behavior is implemented by Line 4. By linearity, the algorithm is correct for nodes representing k-qubit states with $k > a$. This is implemented by Line 5. □

Algorithm 20 Applies a controlled-Z gate to a QMDD node v, with control qubit a and target qubit b. More specifically, given a QMDD node v, representing the state $|v\rangle$, this subroutine returns a QMDD edge e representing $|e\rangle = CZ_a^b\,|v\rangle$. We assume wlog that $a > b$, since $CZ_a^b = CZ_b^a$. Here idx denotes the index of the qubit of v, and Cache denotes a hashmap which maps triples to QMDD nodes. The subroutine ApplyZ applies a Z gate to a given target qubit t.

```
procedure ApplyControlledZ(QMDD node $v$, qubit indices $a, b$)
    Say that node $v$ is $(v_0) \overset{\lambda_0}{\cdots\cdots} (v) \overset{\lambda_1}{-\!\!-\!\!-} (v_1)$
    if the Cache contains the tuple $(v, a, b)$ then return Cache$[v, a, b]$
    else if idx$(v) = a$ then $r :=$ MakeNode($\overset{\lambda_0}{-\!\!-\!\!-}(v_0)$, $\lambda_1 \cdot$ ApplyZ$(v_1, b)$)
    else $r :=$ MakeNode($\lambda_0 \cdot$ ApplyCZ$(v_0, a, b)$, $\lambda_1 \cdot$ ApplyCZ$(v_1, a, b)$)
    Cache$[v, a, b] := r$
    return $r$
procedure ApplyZ(QMDD node $v$, qubit index $t$)
    Say that node $v$ is $(v_0) \overset{\lambda_0}{\cdots\cdots} (v) \overset{\lambda_1}{-\!\!-\!\!-} (v_1)$
    if the Cache contains the tuple $(v, t)$ then return Z-Cache$[v, t]$
    else if idx$(v) = t$ then $r :=$ MakeNode($\overset{\lambda_0}{-\!\!-\!\!-}(v_0)$, $\overset{-\lambda_1}{-\!\!-\!\!-}(v_1)$)
    else $r :=$ MakeNode($\lambda_0 \cdot$ ApplyZ$(v_0, t)$, $\lambda_1 \cdot$ ApplyZ$(v_1, t)$)
    Z-Cache$[v, t] := r$
    return $r$
```

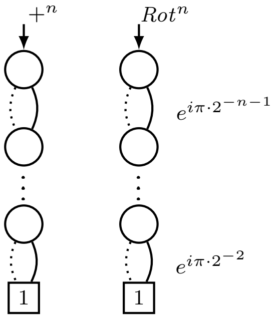


Figure F.1: The states $|+^n\rangle$ and $|Rot^n\rangle$ state as QMDD.

To prove that a single swap operation can explode the LIMDD or QMDD, we first provide two lemmas.

Lemma F.11. For $n \geq 1$, let $|Rot^n\rangle = \bigotimes_{j=1}^{n}\left(|0\rangle + e^{i\pi 2^{-j-1}}|1\rangle\right)$ and $|+^n\rangle = |+\rangle^{\otimes n}$.Then the states $|Rot^n\rangle$ and $|+^n\rangle$ have a linear-size QMDD.

Proof. Figure F.1 provides the QMDD representing both states. ☐

Lemma F.12. The following state has large LIMDD, for any variable order in which the qubit in register B comes after the qubits in register A.

$$|Sum'\rangle = |+\rangle_A^{\otimes n}|0\rangle_B + \bigotimes_{j=1}^{n}(|0\rangle_A + e^{i\pi 2^{-j-1}}|1\rangle_A) \otimes |1\rangle_B \tag{F.1}$$

Proof. The proof is similar to that of Lemma E.4 except that we reason about level 2. ☐

Lemma F.13. QMDD does not support **Swap** in polynomial time.

Proof. Let $|+^n\rangle$ and $|Rot^n\rangle$ be the states from Lemma F.11, and define the following state $|\rho\rangle$ on $n+2$ qubits,

$$|\rho\rangle = |0\rangle|+^n\rangle|0\rangle + |1\rangle|Rot^n\rangle|0\rangle \tag{F.2}$$

Then $|\rho\rangle$ has a small QMDD, of only size $\mathcal{O}(n)$. When we swap the first and last qubits, we obtain a state that includes $|Sum'\rangle$ from Lemma F.12:

$$\text{Swap}_1^n \cdot |\rho\rangle = |0\rangle \otimes (|+^n\rangle|0\rangle + |Rot^n\rangle|1\rangle) \tag{F.3}$$

The QMDD of $\text{Swap}_1^n \cdot |\rho\rangle$ is at least as large as that of $|\text{Sum}\rangle$: First, Wegener [344], Th. 2.4.1, shows that constraining can never increase the DD size, so we can discard the $|0\rangle\otimes$ part (regardless of variable order), as $\text{Swap}_1^n \cdot |\rho\rangle$ is at least as large as $|Sum'\rangle$. Then Lemma F.12 shows that this LIMDD has size at least $2^{\Omega(n)}$ for any variable order. Since, LIMDD is at least as succinct as QMDD (see Figure 5.2), this also holds for QMDD. ☐

Lemma F.14. QMDD does not support **Local** in polynomial time.

Proof. This is implied by Lemma F.13, since **Swap** is a 2-local gate (namely, it involves 2 qubits). ☐

F.3 Easy and hard operations for LIMDD

Lemma F.15. LIMDD supports **Sample**, **Measure**, **Equal** and **X,Y,Z**.

Proof. Vinkhuijzen et al. [337] show that LIMDD supports **Sample**, **Measure**, **Equal** and **X,Y,Z**. ☐

Here we show that LIMDD also support applying a T gate, but does not support **Addition**, H and **Swap**. In this work, we show that computing the fidelity (and hence the inner product, as we can reduce fidelity to inner product) between two states represented by LIMDDs is NP-hard.

Lemma F.16. LIMDD supports Controlled-Z in polynomial time.

Proof. Vinkhuijzen et al. [337] show how to apply any controlled Pauli gate to a state represented by a LIMDD in polynomial time, in the case where the target qubit comes after the control qubit in the variable order of that LIMDD. However, in the case of the controlled-Z, there is no distinction between control and target qubit, since the gate is symmetric. Therefore, their analysis applies to all controlled-Z gates. In fact, inspecting their method, we see that the LIMDD of the resulting state is never larger in size than the LIMDD we started with. ☐

It is known that addition is hard for QMDD (see Table 2 in [113]). For LIMDD, the same was suspected, but not proved in [337]. We show it here by showing that $\langle Z\rangle$-LIMDD does not support addition in polytime.

Lemma F.17. LIMDD does not support **Addition** in polytime.

Proof. Consider the states $|+^n\rangle$ and $|Rot^n\rangle$ as defined in Lemma F.12. Both states have polynomially sized QMDDs as shown in Lemma F.11. Since LIMDD is at least as succinct as QMDD (see Figure 5.2), the LIMDDs representing these states are also small. However, their sum is the state $|\text{Sum}\rangle = |+^n\rangle + |Rot^n\rangle$, which has an exponential-size LIMDD relative to every variable order by Lemma E.4. ☐

Lemma F.18. LIMDD does not support **Hadamard** in polynomial time, and hence neither does it support **Local**.

Proof. Since Hadamard can be used together with measurement (called *conditioning* by Fargier [113]) to realize state addition as explained in Section 5.4, it is also intractable. (Recall also from [344] Th. 2.4.1 that conditioning never increases DD size; this is true in particular for LIMDDs) ☐

We now prove that the fidelity of LIMDDs cannot be computed in polynomial time, under common assumptions of complexity theory.

LIMDD FIDELITY is hard to compute. We show that LIMDD FIDELITY cannot be computed in polynomial time, unless the Exponential Time Hypothesis (ETH) is false. This proof implies that inner product is hard, since fidelity reduces to inner product. Proving hardness of inner product is also a specialized case of the below construction, which does not require our newly defined EOSD problem (see below) but only the well-known hard problem of counting even subgraphs of a certain size (#EVEN SUBGRAPHS).

The proof of LIMDD FIDELITY hardness proceeds in several steps. The starting point is Jerrum and Meeks' result that the problem #EVEN SUBGRAPHS cannot be solved in polynomial time unless ETH is false (Lemma F.19). We introduce a problem we call EVEN ODD SUBGRAPHS DIFFERENCE (EOSD). We give a reduction from #EVEN SUBGRAPHS to EOSD, thus showing that EOSD cannot be solved in polynomial time, under the same assumptions (Lemma F.21). This step is the most technical part of the proof. Finally, we give a reduction form EOSD to LIMDD FIDELITY, thus obtaining the desired result, that LIMDD FIDELITY cannot be computed in polynomial time (to a certain precision), unless ETH is false (Lemma F.20). In this step, we use the fact that LIMDDs can efficiently represent Dicke states and graph states (a type of stabilizer state). Specifically, we will show that computing the fidelity between these states essentially amounts to solving EOSD for the given graph state. Dicke states were first studied by Dicke [101]; see also Bärtschi et al. [32].

We first formally define the three problems above, including computing the fidelity of two LIMDDs. We will need the following terminology for graphs. For an undirected graph $G = (V, E)$ and a set of vertices $S \subseteq V$, we denote by $G[S]$ the subgraph induced by S. If $|S| = k$, then we say that $G[S]$ is a k-induced subgraph, and we say that it is an *even* (resp. *odd*) subgraph if $G[S]$ has an even (resp. odd) number of edges. We let $e(G,k)$ (resp. $o(G,k)$) denote the number of even (resp. odd) k-induced subgraphs of G.

LIMDD FIDELITY.

Input: Two LIMDDs, representing the states $|\varphi\rangle, |\psi\rangle$

Output: The value $|\langle\varphi|\psi\rangle|^2$ to $2n$ bits of precision.

#EVEN SUBGRAPHS

Input: A graph $G = (V, E)$, an an integer k

Output: The value $e(G,k)$.

EVEN ODD SUBGRAPH DIFFERENCE (EOSD).

Input: A graph $G = (V, E)$, and an integer k.

Output: The value $|e(G,k) - o(G,k)|$, i.e., the absolute value of the difference between the number of even and odd induced k-subgraphs of G.

Lemma F.19 (Jerrum and Meeks [169])**.** If #EVEN SUBGRAPHS is polytime, then ETH is false.

Proof. Jerrum and Meeks [169] showed that counting the number of even induced subgraphs with k vertices is #W[1]-hard. Consequently, there is no algorithm running in time $poly(n)$ (independent of k) unless the exponential time hypothesis fails. ☐

Lemma F.20. There is no polynomial-time algorithm for LIMDD FIDELITY, i.e., for computing fidelity between two LIMDDs to $2n$ bits of precision, unless the Exponential Time Hypothesis (ETH) fails.

Proof. Suppose there was such a polynomial-time algorithm, running in time $\mathcal{O}(n^c)$ for some constant $c \geq 1$. We will show that then EOSD can be solved in time $\mathcal{O}(n^c)$ (independent of k), by giving a reduction from EOSD to LIMDD FIDELITY. From Lemma F.21, it would then follow that ETH is false.

The reduction from EOSD to LIMDD FIDELITY is as follows.

Let G be an input graph on n vertices V and $0 \leq k \leq n$ an integer. Let $|G\rangle$ be the graph state corresponding to G [324], so that

$$|G\rangle = \frac{1}{2^{n/2}} \sum_{S \subseteq V} (-1)^{|G[S]|} |S\rangle \tag{F.4}$$

where $|G[S]|$ denotes the number of edges in the S-induced subgraph of G, and $|S\rangle$ denotes the computational-basis state $|S\rangle = |x_1\rangle \otimes |x_2\rangle \otimes ... \otimes |x_n\rangle$ with $x_j = 1$ if $j \in S$ and $x_j = 0$ otherwise. Let $|D_n^k\rangle$ be the Dicke state [101].

$$|D_n^k\rangle = \frac{1}{\sqrt{\binom{n}{k}}} \sum_{\vec{x}\in\{0,1\}^n \text{ with } |\vec{x}|=k} |\vec{x}\rangle \tag{F.5}$$

Both these states have small LIMDDs:

Dicke state. Bryant [67] gives a construction for BDDs to represent the function $f_k\colon \{0,1\}^n \to \{0,1\}$, with $f_k(x) = 1$ iff $|x| = k$. This is precisely the amplitude function of the Dicke state $|D_n^k\rangle$ (up to a factor $1/\sqrt{\binom{n}{k}}$). This construction also works for LIMDDs, by simply setting all the edge labels to the identity, and using root label $1/\sqrt{\binom{n}{k}} \cdot \mathbb{I}^{\otimes n}$.

Graph state. Vinkhuijzen et al. [337] show how to efficiently construct a LIMDD for any graph state.

It is straightforward to verify that the fidelity between $|D_n^k\rangle$ and $|G\rangle$ is related to the subgraphs of G, as follows,

$$\langle D_n^k|G\rangle = \frac{1}{\sqrt{\binom{n}{k}2^n}} \sum_{S\subseteq V:|S|=k} (-1)^{|G[S]|} = \frac{1}{\sqrt{\binom{n}{k}2^n}} \left(e(G,k) - o(G,k)\right) \tag{F.6}$$

Hence,

$$\underbrace{|e(G,k) - o(G,k)|}_{\text{solution to EOSD}} = \sqrt{\binom{n}{k} 2^n \underbrace{|\langle D_n^k|G\rangle|^2}_{\text{Fidelity}}} \tag{F.7}$$

Since $|\langle D_n^k|G\rangle|^2$ denotes the fidelity between $|D_n^k\rangle$ and $|G\rangle$, and $|e(G,k) - o(G,k)|$ denotes the quantity asked for by the EOSD problem, this completes the reduction. The overhead of constructing the LIMDDs from the description of the Dicke and graph states takes linear time in the size of the resulting LIMDD. So, if the fidelity of two LIMDDs is computed in polynomial time, say, in time $\mathcal{O}(n^c)$, then also the quantity $|e(G,k) - o(G,k)|$ is computed in time $\mathcal{O}(n^c)$; thus, EOSD is solved in time $\mathcal{O}(n^c)$. Lastly, we address the number of bits of precision required. In order to exactly compute the integer $|e(G,k) - o(G,k)|$, it is necessary to compute the fidelity $|\langle D_n^k|G\rangle|^2$ with a precision of at least one part in $\binom{n}{k}2^n$. Put another way, the required number of bits of precision is $\log_2(\binom{n}{k}\cdot 2^n) \leq \log_2(2^n\cdot 2^n) = 2n$. Summarizing, computing the fidelity of (the states represented by) two LIMDDs representing a graph state and a Dicke state, to $2n$ bits of precision, is not possible in polynomial time, unless ETH fails. ☐

Lemma F.21. There is no polynomial-time algorithm for EOSD, unless ETH is false.

Proof. We provide an efficient reduction (in Algorithm 21) from #EVEN SUBGRAPHS: the problem, on input an undirected graph G and a parameter $k \in \{0, 1, 2, ..., |V|\}$, of computing the number of k-vertex induced subgraphs which have an even number of edges. It follows that, if EOSD can be computed in polynomial time, then Algorithm 21, which computes #EVEN SUBGRAPHS, also runs in polynomial time. Jerrum and Meeks [169] show that #EVEN-SUBGRAPHS cannot be computed in polynomial time unless ETH is false (Lemma F.19). Therefore, if EOSD could be computed in polynomial time, then ETH would be false.

The algorithm CountEvenSubgraphs (Algorithm 21) takes as parameters a graph G and an integer $k \geq 0$, and outputs $e(G,k)$, the number of even k-induced subgraphs of G, thus solving #EVEN SUBGRAPHS. This algorithm uses at most $2n$ invocations of a subroutine EvenOddSubgraphsDifference; therefore, if the subroutine EvenOddSubgraphsDifference runs in polynomial time, then so does CountEvenSubgraphs.

Let us briefly sketch the idea behind the algorithm, before we give a formal proof of correctness. First, we know that $e(G,k)+o(G,k) = \binom{n}{k}$, since each subgraph is either even or odd, and G has $\binom{n}{k}$ different k-induced subgraphs in total. Thus, if we knew the (possibly negative) difference $\zeta_k = e(G,k) - o(G,k)$, then we know the sum and difference of $e(G,k)$ and $o(G,k)$, so we could compute the desired value $e(G,k) = \frac{1}{2}(\binom{n}{k} + \zeta_k)$. Unfortunately, EvenOddSubgraphsDifference only tells us the absolute value, $|\zeta_k|$. Fortunately, we know that $e(G,0) = 1$ and $o(G,0) = 0$, so $\zeta_0 = 1 - 0 = 1$ (namely, there is only one induced subgraph with 0 vertices, and it has 0 edges, which is even). We now bootstrap our way up, computing ζ_j for $j = 1, \ldots, k$ using the previously known results. The key ingredient is that, by adding isolated vertices to the graph and querying EvenOddSubgraphsDifference on this new graph, we can discover the the absolute difference $|\zeta_j + \zeta_{j-1}|$, which allows us to compute the values ζ_j.

Correctness of the algorithm. We now prove that the algorithm CountEven-

SUBGRAPHS outputs the correct value. Let (G, k) be the input to the algorithm. For $j = 0, \ldots, k$, let $\zeta_j = e(G, j) - o(G, j)$. We will show that, for each $j = 1, \ldots, k$, the algorithm sets the variable d_j to the value ζ_j in the j-th iteration of the for-loop. The proof is by induction on j. In the induction hypothesis, we include also that the variable ℓ is always the largest value below j satisfying $\zeta_\ell \neq 0$ as in Equation F.8 (this value is well-defined, since $1 = \zeta_0 \neq 0$, so we have $0 \leq \ell < j$).

$$\ell = \max\{0 \leq \ell < j \mid \zeta_\ell \neq 0\} \tag{F.8}$$

For the base case, where $j = 0$, it suffices to note that there is only one set with zero vertices – the empty set – which induces the empty graph, which contains an even number of edges. Therefore, $\zeta_0 = 1$, which the algorithm sets on Line 2. Finally, ℓ is correctly set to 0.

For the induction case $j \geq 1$, the variables d_t have been set to $d_t = \zeta_t$ for $t = 0, \ldots, j-1$ and ℓ satisfies Equation F.8 from the induction hypothesis. Consequently, we have $\zeta_{\ell+1} = \cdots = \zeta_{j-1} = 0$. If $\zeta_j = 0$, then the algorithm sets $q := |\zeta_j| = |0| = 0$ on Line 5, so the algorithm sets d_j correctly on Line 7, and correctly leaves ℓ untouched (ℓ remains unchanged from the $j-1$-th to the j-th iteration). Otherwise, if $\zeta_j \neq 0$,

Algorithm 21 An algorithm which computes the number of even k-induced subgraphs using at most $2n$ calls to a subroutine EVENODDSUBGRAPHSDIFFERENCE, which returns $|e(G, k) - o(G, k)|$ on input (G, k).

```
1:  procedure CountEvenSubgraphs($G = (V, E), k$)
    Output: The number of even induced subgraphs of $G$ with $k$ vertices
2:      $d_0 := 1$                                        ▷ $d$ is an array of $k + 1$ integers
3:      $\ell := 0$                                       ▷ Last iteration when $\zeta_j = 1$
4:      for $j := 1, \ldots, k$ do
5:          $q :=$ EvenOddSubgraphsDifference($G, j$)
6:          if $q = 0$ then                               ▷ There are equally many even as odd subgraphs
7:              $d_j := 0$
8:          else              ▷ Else we have to figure out whether there more even or odd subgraphs:
9:              $G' := (V \cup \{v'_1, \ldots, v'_{j-\ell}\}, E)$      ▷ Add $j - \ell$ new isolated vertices
10:             $p :=$ EvenOddSubgraphsDifference($G', j$)
11:             $d_j := \begin{cases} q & \text{if } |d_\ell + q| = p \\ -q & \text{if } |d_\ell - q| = p \end{cases}$
12:             $\ell := j$                ▷ Since iteration $j$ is the latest iteration having $\zeta_j = 1$
13:     return $\frac{1}{2}\left(\binom{n}{k} + d_k\right)$
```

the algorithm adds $j-\ell$ new, isolated vertices to G, obtaining the new graph $G' = (V_G \cup \{v'_1, \ldots, v'_{j-\ell}\}, E_G)$. On Line 10, it computes the value $|e(G', j) - o(G', j)|$ of this graph. Since this expression also sums over induced subgraphs of G' that contain isolated vertices, this value can be expressed as follows:

$$p := |e(G', j) - o(G', j)| \tag{F.9}$$

$$= \left|\sum_{a=\ell}^{j} \binom{j-\ell}{j-a} (e(G,a) - o(G,a))\right| = \left|\sum_{a=\ell}^{j} \binom{j-\ell}{j-a} \zeta_a\right| \tag{F.10}$$

$$= \left|\binom{j-\ell}{j-\ell}\zeta_\ell + \binom{j-\ell}{0}\zeta_j\right| = |\zeta_\ell + \zeta_j| \tag{F.11}$$

We noted that $\zeta_{\ell+1} = \cdots = \zeta_{j-1} = 0$; therefore, these terms vanish from the summation (step from Equation F.10 to Equation F.11), so that only $p = |\zeta_j + \zeta_\ell|$ remains. Since we now know the values of $\zeta_\ell, |\zeta_j|$ and $|\zeta_j + \zeta_\ell|$ and since $\zeta_\ell \neq 0$, we can infer the value of ζ_j, which is done on Line 11. We conclude that each variable d_j is correctly set to ζ_j, concluding the proof by induction. Also, since $\zeta_j \neq 0$, ℓ is correctly set to j.

Lastly, we show that the value returned by the algorithm is indeed the number $e(G,k)$. Suppose that $d_k = \zeta_k = e(G,k) - o(G,k)$. We know that $e(G,k) + o(G,k) = \binom{n}{k}$. That is, we know both the sum and the difference of $e(G,k), o(G,k)$; therefore we can compute them both. By adding these two equations and solving for $e(G,k)$, we obtain $e(G,k) = \frac{1}{2}\left(\binom{n}{k} + d_k\right)$, which is the value returned by the algorithm. ☐

Lemma F.22. LIMDD supports the T-gate.

Proof. Algorithm 22 applies an arbitrary diagonal gate $D = \begin{bmatrix} \rho & 0 \\ 0 & \omega \end{bmatrix}$ to a state represented by a LIMDD. To apply a T-gate, one calls the algorithm with $D = T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$. We now show that the algorithm runs in polynomial time. First, since each recursive call takes $\mathcal{O}(1)$ time, for the purposes of estimating runtime it suffices to count the number of recursive calls. The cache stores all tuples of nodes and matrices with which the algorithm is called, so for the purposes of estimating the runtime it suffices to count the number of *distinct* recursive calls. To this end, we note that the recursive calls to the algorithm only receive two different matrices, namely $\begin{bmatrix} \rho & 0 \\ 0 & \omega \end{bmatrix}$ and $\begin{bmatrix} \omega & 0 \\ 0 & \rho \end{bmatrix}$. The nodes that are passed as argument v are nodes that are already in the diagram. Therefore, if the diagram contains m nodes, then at most $2m$ distinct recursive calls are made. We conclude that the runtime is polynomial (indeed, linear), in the size of the diagram. ☐

Algorithm 22 Applies a diagonal gate D to qubit k of a state represented by a LIMDD.

```
procedure ApplyDiagGate(LIMDD Node $v = \textcircled{v_0} \overset{\lambda_0 A_0}{\cdots\cdots} \bigcirc \overset{\lambda_1 A_1}{\text{———}} \textcircled{v_1}$, gate $D$, qubit $k$)
with $D = \left[\begin{smallmatrix}\rho & 0\\ 0 & \omega\end{smallmatrix}\right]$
Node $v$ represents a state on $n$ qubits
    if Cache contains the tuple $(v, D)$ then return Cache$[v, D]$
    else if $k = n$ then
        return $\textcircled{v_0} \overset{\rho\lambda_0 A_0}{\cdots\cdots} \bigcirc \overset{\omega\lambda_1 A_1}{\text{———}} \textcircled{v_1}$
    else
        gate $E := \left[\begin{smallmatrix}\omega & 0\\ 0 & \rho\end{smallmatrix}\right]$
        for $i = 0, 1$ do
            gate $F_i := \begin{cases} D & \text{if } A_i^k \in \{I, Z\} \\ E & \text{if } A_i^k \in \{X, Y\} \end{cases}$    ▷ Here $A_i^k$ denotes the $k$-th qubit of the Pauli operator $A^i$
            Node $u_i :=$ ApplyTGateToLIMDD$(v_0, F_i, k)$
        Node $r := \textcircled{u_0} \overset{\lambda_0 A_0}{\cdots\cdots} \bigcirc \overset{\lambda_1 A_1}{\text{———}} \textcircled{u_1}$
        Cache$[v, D] := r$
        return $r$
```

F.4 Easy operations for MPS

Vidal [335] shows that MPS supports efficient application of a single one-qubit gate or two-qubit gate on consecutive qubits, which includes **X,Y,Z**, **Hadamard**, **T**. This extends to any two-qubit operation on any pair of qubits [262], particularly including **Swap** and **CZ** gates. These algorithms are extendable to k-local gates on adjacent qubits, which does not increase the largest matrix dimension D to more than D^k. The algorithm consists of merging the k tensors (the j-th tensor combines the two matrices A_j^0 and A_j^1) into a single large one, applying the gate to the large tensor, followed by splitting the tensor again into k matrices A_j^0 and A_j^1 again by use of the singular-value decomposition (for details on the merging and splitting see e.g. Dang et al. [93]). The largest matrix dimension during this process does not increase above D^k. Using **Swap** gates, one thus implements **Local** on any qubits. We give a direct proof of the support for addition below (Lemma F.23).

Orus [248] gives an accessible exposition of TN, of which MPS is a special case. He explains how to compute the inner product in polynomial time. Thus, MPS also supports **Measure**. **Sample** can be done by a Markov Chain Monte Carlo approach, invoking **Measure** as subroutine. Since inner product is supported, so is **Equal**: MPS M and M' are equivalent iff $\frac{|\langle M|M'\rangle|^2}{\langle M|M\rangle\cdot\langle M'|M'\rangle} = 1$.

Lemma F.23. MPS supports addition in polynomial time.

Proof. Let A, B be MPSs. Then a new MPS C representing $|C\rangle = |A\rangle + |B\rangle$ can be efficiently constructed as follows, for $x = 0, 1$ and $j = 2, \ldots, n-1$:

$$C_n^x = \begin{bmatrix} A_n^x & B_n^x \end{bmatrix} \qquad C_j^x = \begin{bmatrix} A_j^x & 0 \\ 0 & B_j^x \end{bmatrix} \qquad C_1^x = \begin{bmatrix} A_1^x \\ B_1^x \end{bmatrix} \tag{F.12}$$

□

F.5 Easy and hard operations for RBM

Jonsson et al. [174] show that RBM supports Pauli gates, the controlled-Z gate and the T-gate (and, in fact, arbitrary phase gates). There is at the moment no efficient exact algorithm for the **Hadamard** gate, which would make the list of supported gates universal. Hence there is at the moment no exact efficient algorithm for **Local** either. **Sample** is supported for any n-qubit RBM M, see e.g. Appendix B of [174] and references therein, by performing a Markov Chain Monte Carlo algorithm (e.g. Metropolis algorithm) where the Markov Chain state space consists of all bit strings $x \in \{0,1\}^n$, and the corresponding unnormalized probability $|\langle x|M\rangle|^2$ of each state is efficiently computed using Equation 5.1. No exact algorithm for **Equal** is known (in fact, the related problem of identity testing when one only has sampling access to one of the two RBMs is already computationally hard [52]). Although no exact algorithm for **InnerProd** is known, it can be approximated using **Sample** as subroutine (see e.g. Wu et al. [354]). Furthermore, **Measure** can be approximated by computing the normalization factor $1/\langle M|M\rangle$ using the (exact or approximate) algorithm for **InnerProd**, while the relative outcome probabilities are defined in Equation 5.1.

Lemma F.24. RBM supports **Swap**.

Proof. In order to effect a swap between qubits q_1 and q_2, we simply exchange rows q_1 and q_2 in the matrix W and the vector $\vec{\alpha}$, obtaining W' and $\vec{\alpha}'$. Then $\mathcal{M}' = (\vec{\alpha}', \vec{\beta}, W', m)$ has $|\mathcal{M}'\rangle = \textsc{Swap}(q_1, q_2) \cdot |\mathcal{M}\rangle$. □

Torlai et al. [315] note that RBMs can exactly represent Dicke states. In Lemma F.25, we give another construction of succinct RBMs for Dicke states, where the number of hidden nodes grows linearly with the number of visible nodes.

Lemma F.25. An RBM can exactly represent any Dicke state, using only $2n$ hidden nodes.

Proof. We will construct an RBM with $2n$ hidden nodes representing $|D_n^k\rangle$.

For each $j \in \{0, 1, \dots, n\} \setminus \{k\}$, our construction will use two hidden nodes. Fix such a j. Then the first hidden node is connected to each visible node with weight $i\pi/n$, and has bias $b_j = i\pi(1 - j/n)$. The second hidden node is connected to each visible node with weight $-i\pi/n$ and has bias $b_j = -i\pi(1 - j/n)$. Since the weights on all edges incident to a given hidden node are the same, the term it contributes depends only on the weight of the input (i.e., the number of zeroes and ones). Thus, these two nodes contribute a multiplicative factor $(1 + e^{i\pi(1+|x|/n+j/n)})$ and $(1 + e^{-i\pi(1+|x|/n-j/n)})$, respectively. Multiplying these together, the two terms collectively contribute a multiplicative term of $2+2\cos(\pi(1+|x|/n-j/n) = 2-2\cos(\pi(|x|-j)/n)$, which is 0 iff $|x| = j$ and nonzero otherwise. Let a be the constant $a = \prod_{j=0,j\neq k}^{n} 2 - 2\cos(\pi(k-j)/n)$, i.e., the product of all terms when $|x| = k$. Then the RBM represents the following state unnormalized function $\Psi\colon \{0,1\}^n \to \mathbb{C}$, which we then normalize to obtain the state $|\Psi\rangle$:

$$\Psi(x) = \begin{cases} a & \text{if } |x| = k \\ 0 & \text{otherwise} \end{cases} \qquad\qquad |\Psi\rangle = \frac{1}{a\sqrt{\binom{n}{k}}} \sum_x \Psi(x)\,|x\rangle \tag{F.13}$$

The normalized state $|\Psi\rangle$ represents exactly the Dicke state $|D_n^k\rangle$, since its amplitudes are equal to $1/\sqrt{\binom{n}{k}}$ when $|x| = k$ and zero otherwise. □

Since RBM can succinctly represent both Dicke states (Lemma F.25) and graph states, a subset of stabilizer states [365], the proof for hardness of LIMDD FIDELITY (Lemma F.20) is also applicable to RBM.

Corollary F.1. There is no polynomial-time algorithm for RBM FIDELITY, i.e., for computing fidelity between two RBM to $2n$ bits of precision, unless the Exponential Time Hypothesis (ETH) fails.

Appendix G

Proofs of Section 5.5

Section G.1 proves that our rapidity definition is a preorder and is equivalent to the one given by Lai et al. [194] for canonical data structures.

Section G.2 provides the proof for the sufficient condition for rapidity. Section G.3-G.5 apply this sufficient condition to the data structures studied in this work.

G.1 Rapidity is a preorder and generalizes earlier definitions

We now show that rapidity is a preorder over data structures and that the definition of Lai et al. [194] can be considered a special case for canonical data structures. For convenience, we restate the definition of rapidity.

Definition 5.3 (Rapidity for non-canonical data structures)**.** Let D_1, D_2 be two data structures and consider some c-ary operation OP on these data structures. In the below, ALG_1 (ALG_2) is an algorithm implementing OP for D_1 (D_2).

(a) We say that ALG_1 is *at most as rapid as* ALG_2 iff there exists a polynomial p such that for each input $\varphi = (\varphi_1, \ldots, \varphi_c)$ there exists an equivalent input $\psi = (\psi_1, \ldots, \psi_c)$, i.e., with $|\varphi_j\rangle = |\psi_j\rangle$ for $j = 1 \ldots c$, for which $\mathit{time}(ALG_2, \psi) \leq p\left(\mathit{time}(ALG_1, \varphi)\right)$. We say that ALG_2 is *at least as rapid as* ALG_1.

(b) We say that $OP(D_1)$ is *at most as rapid as* $OP(D_2)$ if for each algorithm ALG_1 performing $OP(D_1)$, there is an algorithm ALG_2 performing $OP(D_2)$ such that ALG_1 is at most as rapid as ALG_2.

Theorem 5.4. Rapidity is a preorder over data structures.

Proof. We first show that rapidity is reflexive, next we show that it is transitive.

Rapidity is reflexive. It suffices to show that rapidity is a reflexive relation on algorithms performing a given operation. Let D be a data structure, OP an operation and ALG an algorithm performing $OP(D)$. Then ALG is at most as rapid as itself if there exists a polynomial p such that for each input φ there exists an equivalent input ψ with $time(ALG, \varphi) \leq p(ALG, \psi)$. We may choose the polynomial $p(x) = x$, and we may choose $\psi := \varphi$. Then the statement reduces to the trivial statement $time(ALG, \varphi) = time(ALG, \psi) = p(ALG, \psi)$.

Rapidity is transitive. It suffices to show that rapidity is a transitive relation on algorithms. To this end, let D_1, D_2, D_3 be data structures, OP an operation and ALG_1, ALG_2, ALG_3 algorithms performing $OP(D_1), OP(D_2), OP(D_3)$, respectively. Suppose that ALG_1 is at most as rapid as ALG_2 and ALG_2 is at most as rapid as ALG_3. We will show that ALG_1 is at most as rapid as ALG_3. By the assumptions above, there are polynomials p and q such that (i) for each input φ there exists an equivalent input ψ such that $time(ALG_2, \psi) \leq p(time(ALG_1, \varphi))$; and (ii) for each input ψ there exists an equivalent input γ such that $time(ALG_3, \gamma) \leq q(time(ALG_2, \psi))$.

Put together, for every input φ there exist equivalent inputs ψ and γ such that $time(ALG_3, \gamma) \leq q(time(ALG_2, \psi)) \leq q(p(time(ALG_1, \varphi)))$. Letting the polynomial $\ell(x) = q(p(x))$, we obtain that for every φ there exists an equivalent γ such that $time(ALG_3, \gamma) \leq \ell(ALG_1, \varphi)$. □

We note that an alternative definition of rapidity [194], which always allows ALG_2 to read its input by requiring $\text{time}(ALG_2, y) \leq p(\text{time}(ALG_1, x) + |y|)$ instead of $\text{time}(ALG_2, y) \leq p(\text{time}(ALG_1, x))$, is not transitive for query operations:

Consider the data structure Padded QMDD, (PQMDD) which is just a QMDD, except that a string of 2^{2^n} "0"'s have been concatenated to the end of the QMDD representation, where n is the number of qubits.

Under the alternative rapidity relation $\geq_r^{\text{alt}}$, both ADD and QMDD are at least as rapid as PQMDD, because the ADD algorithm is allowed to run for $poly(2^{2^n})$ time. But PQMDD is also at least as rapid as QMDD, because algorithms for PQMDD don't need to read the whole 2^{2^n}-length input — they only read the QMDD at the beginning of the string. Put together, this leads to:

$$\mathsf{ADD} \geq_r^{\text{alt}} \mathsf{PQMDD} \geq_r^{\text{alt}} \mathsf{QMDD} \quad \text{and} \quad \mathsf{ADD} \not\geq_r^{\text{alt}} \mathsf{QMDD}.$$

Next, we show that our definition of rapidity is equivalent to Lai et al.'s definition of rapidity in the case when both data structures are canonical and we restrict our attention to only those algorithms which run in time at least m where m is the size of the input. For convenience, we restate Lai et al.'s definition here.

Definition G.1 (Rapidity for canonical data structures [194])**.** A c-ary operation OP on a canonical language L_1 is *at most as rapid as* OP on another canonical language L_2, iff for each algorithm ALG performing OP on L_1 there exists some polynomial p and some algorithm ALG_2 performing OP on L_2 such that for every valid input $(\varphi_1, \ldots, \varphi_c, \alpha)$ of OP on L_1 and every valid input $(\psi_1, \ldots, \psi_c, \alpha)$ of OP on L_2 satisfying $\varphi_i \equiv \psi_i$ $(1 \leq i \leq c)$, $ALG_2(\psi_1, \ldots, \psi_c, \alpha)$ can be done in time $p(t + |\varphi_1| + \cdots + |\varphi_c| + |\alpha|)$, where α is any element of supplementary information and t is the running time of $ALG(\varphi_1, \ldots, \varphi_c, \alpha)$.

Lai et al. use several minor differences in notation. First, they speak of *valid* inputs (because they consider data structures which cannot represent all objects), whereas we do not; they use an element of supplementary information α as part of the input, whereas we omit such an element; they write $\varphi_i \equiv \psi_i$ where we write $|\varphi_i\rangle = |\psi_i\rangle$; lastly they speak of a *language* whereas we speak of a *data structure*. Since these differences between the notation are inconsequential, it will be convenient to rephrase the definition of Lai et al. using the notation of this work, as follows:

Definition G.2 (Rapidity of canonical data structures, rephrased)**.** In the following, ALG_1, ALG_2 are algorithms which perform OP on canonical data structures D_1, D_2, respectively.

(a) An algorithm ALG_1 is *at most as rapid as* an algorithm ALG_2 iff there is a polynomial p such that for each input φ and for each equivalent input ψ, it holds that $time(ALG_2, \psi) \leq p(time(ALG_1, \varphi) + |\varphi|)$.

(b) A canonical data structure D_1 is *at most as rapid as* a canonical data structure D_2 for an operation OP if for each algorithm ALG_1 performing OP on D_1 there is an algorithm ALG_2 performing OP on D_2 such that ALG_1 is at most as rapid as ALG_2.

Lemma G.1. Definition 5.3 is equivalent to the definition of [194] (Definition G.2) in the case when two data structures D_1, D_2 are both canonical and where we restrict our attention to algorithms whose runtime is at least m, where m is the size of the input.

Proof. Let D_1, D_2 be two canonical data structures. We will show that D_1 is at most as rapid as D_2 according to Definition 5.3 if and only if the same is true according tot Definition G.2. Since items 5.3.(b) and G.2.(b) are equivalent, it suffices to show that the two definitions are equivalent for *algorithms* rather than *data structures*. That is, we will show that an algorithm ALG_1 is at most as rapid as ALG_2 according to Definition 5.3 if and only if the same is true according to Definition G.2.

Abusing notation, we write $|(\varphi_1, \ldots, \varphi_c)|$ instead of $|\varphi_1| + \ldots + |\varphi_c|$, etc. In this proof, we will assume without loss of generality that all polynomials p are monotonically increasing (i.e., $p(x) \leq p(y)$ if $x \leq y$). Namely, if p is a polynomial which does not monotonically increase, then use instead the polynomial $p'(x) = p(x) + x^k$ for sufficiently large k.

Direction *if*. Let ALG_1, ALG_2 be algorithms performing OP on canonical data structures D_1, D_2, respectively, such that ALG_1 is at most as rapid as ALG_2 according to Definition G.2. Then there is a polynomial p such that $time(ALG_2, \psi) \leq p(time(ALG_1, \varphi) + |\varphi|)$ for all equivalent inputs φ, ψ. Since the data structures D_1, D_2 can represent all quantum state vectors, there certainly *exists* an equivalent ψ to any φ; indeed, since D_2 is canonical, there is a unique such instance ψ. Since we restrict our attention to algorithms with runtime at least m where m is the size of the input, we get that $|\varphi| \leq time(ALG_1, \varphi)$, so $p(time(ALG_1, \varphi) + |\varphi|) \leq p(2 \cdot time(ALG_1, \varphi))$.

Therefore, let $q(x) = p(2x)$. Now we get that, for every input φ, there exists an equivalent input ψ such that $time(ALG_2, \psi) \leq q(ALG_1, \varphi)$. Therefore, ALG_1 is at most as rapid as ALG_2 according to Definition 5.3.

Direction *only if*. Suppose that ALG_1 is at most as rapid as ALG_2 according to Definition 5.3. Then there is a polynomial p such that for each input φ, there is an equivalent input ψ such that $time(ALG_2, \psi) \leq p(time(ALG_1, \varphi))$. Using the monotonicity of p which we assume without loss of generality, we get that $p(time(ALG_1, \varphi)) \leq p(time(ALG_1, \varphi) + |\varphi|)$. Lastly, since D_2 is canonical, any instance ψ which is equivalent to φ must be the *only* input instance that is equivalent to φ. Therefore, we obtain that there exists a polynomial p such that for each input φ and for all equivalent inputs ψ (i.e., for the unique equivalent instance ψ of D_2), it holds that $time(ALG_2, \psi) \leq p(time(ALG_1, \varphi) + |\varphi|)$. Therefore, ALG_1 is at most as rapid as ALG_2 according to Definition G.2. □

G.2 A Sufficient Condition for Rapidity

Here, we prove Theorem 5.5, which we restate below.

Theorem 5.5 (A sufficient condition for rapidity)**.** Let D_1, D_2 be data structures with $D_1 \preceq_s D_2$ and OP a c-ary operation. Suppose that,

A1 $OP(D_2)$ requires time $\Omega(m)$ where m is the sum of the sizes of the operands; and

A2 for each algorithm ALG implementing $OP(D_2)$, there is a runtime monotonic algorithm ALG^{rm}, implementing the same operation $OP(D_2)$, which is at least as rapid as ALG; and

A3 there exists a transformation from D_1 to D_2 which is (i) weakly minimizing and (ii) runs in time polynomial in the output size (i.e, in time $\mathsf{poly}(|\psi|)$ for transformation output $\psi \in D_2$); and

A4 if OP is a manipulation operation (as opposed to a query), then there also exists a polynomial time transformation from D_2 to D_1 (polynomial time in the input size, i.e, in $|\rho|$ for transformation input $\rho \in D_2$).

Then D_1 is at least as rapid as D_2 for operation OP.

Proof. We prove the theorem for $c = 1$. This can be easily extended to the case with multiple operands by treating the operands point-wise and summing their sizes. We show that $OP(D_2)$ is at most as rapid as $OP(D_1)$, assuming that the conditions in Theorem 5.5 hold. (Note that this swaps the roles of D_1 and D_2 relative to Definition 5.3). In this proof, we will assume without loss of generality that all polynomials p are monotone, i.e., if $x \leq y$ then $p(x) \leq p(y)$.

We prove the theorem for a manipulation operation OP. The proof for a query operation OP follows as a special case, which we treat at the end of the proof.

Let ALG_2 be an $\Omega(m)$ algorithm implementing $OP(D_2)$. By A2, we may assume without loss of generality that ALG_2 is runtime monotonic. Let $f\colon D_1 \to D_2$ be the polynomial-time weakly minimizing transformation (A3), and $g\colon D_2 \to D_1$ the polynomial-time transformation in the other direction satisfying the criteria in A4.

We set $ALG_1 = f \circ ALG_2 \circ g$, i.e., ALG_1 is as follows.

```
procedure $ALG_1(\varphi)$
    $\psi := f(\varphi)$
    $\rho := ALG_2(\psi)$
    return $g(\rho)$
```

ALG_1 is *complete* (i.e., works on all inputs), since f, g and ALG_2 are. The remainder of the proof shows that ALG_2 is at most as rapid as ALG_1, i.e., there exists a polynomial p such that for all operands $\psi \in D_2$, there exists in input $\varphi \in D_1$ with $|\varphi\rangle = |\psi\rangle$ for which $time(ALG_1, \varphi) \leq p\,(time(ALG_2, \psi))$.

Let $\psi \in D_2$. We take $\varphi \in D_1$ such that $|\varphi\rangle = |\psi\rangle$ and $|\varphi| \leq s(|\psi|)$ for the polynomial s ensuring the succinctness relation $D_1 \preceq_s D_2$. Such a φ exists, because D_1 is more succinct than D_2.

It remains to show that $\exists p\colon time(ALG_1, \varphi) \leq p\,(time(ALG_2, \psi))$, where p is independent of φ and ψ. To this end, we can express the time required by ALG_1 by summing the runtimes of its three steps as follows.

$$time(ALG_1, \varphi) = time(f, \varphi) + time(ALG_2, f(\varphi)) + time(g, ALG_2(f(\varphi))) \tag{G.1}$$

It now suffices to prove that each summand of Equation G.1 is polynomial in the runtime of $ALG_2(\psi)$.

1. We show $time(f, \varphi) \leq \mathsf{poly}(time(ALG_2, \psi))$. Since f runs in polynomial time in its output (A3) and $|\varphi| \leq s(|\psi|)$ (see above), we have $time(f, \varphi) \leq \mathsf{poly}(|f(\varphi)|)$. Let t be the polynomial such that $time(f, \varphi) \leq t(|f(\varphi)|)$. Since f is weakly minimizing (A3), it is guaranteed that $|f(\varphi)| \leq m(|\psi|)$ for some polynomial m. Lastly, by A1, we have $|\psi| = \mathcal{O}(time(ALG_2^{rm}, \psi))$, so $|\psi| \leq k(time(ALG_2, \psi))$ for some polynomial k. Put together, we have $time(f, \varphi) \leq t(|f(\varphi)|) \leq t(m(|\psi|)) \leq$

$t(m(k(\mathit{time}(ALG_2, \psi))))$, which proves the claim.

2. We show $\mathit{time}(ALG_2, f(\varphi)) \leq \mathsf{poly}(\mathit{time}(ALG_2, \psi))$. Because f is a weakly minimizing transformation (A3), we have $|f(\varphi)| \leq s(|\psi|)$ for some s. Since ALG_2 is runtime monotonic (A2), and because $|f(\varphi)| \leq s(|\psi|)$, we have $\mathit{time}(ALG_2, f(\varphi)) \leq t(ALG_2, \psi)$ for some t, which proves the claim.

3. We show $\mathit{time}(g, ALG_2(f(\varphi))) \leq \mathsf{poly}(\mathit{time}(ALG_2, \psi))$. Since g runs in time polynomial in the input (A4); and the input to g is $ALG_2(f(\varphi))$, we have $\mathit{time}(g, ALG_2(f(\varphi))) \leq p(|ALG_2(f(\varphi))|)$ for some polynomial p. Next, we have trivially $\mathit{time}(ALG_2, f(\varphi)) \geq |ALG_2(f(\varphi))|$, since the time ALG_2 spends writing the output is included in the total time, thus we obtain $\mathit{time}(g, ALG_2(f(\varphi))) \leq p(\mathit{time}(ALG_2, f(\varphi)))$. As we have seen above in item 2, $\mathit{time}(ALG_2, f(\varphi)) \leq t(\mathit{time}(ALG_2, \psi))$ for some polynomial t. Putting this together, we obtain $\mathit{time}(g, ALG_2(f(\varphi))) \leq p(t(\mathit{time}(ALG_2, \psi)))$, which proves the claim.

This proves the theorem for the case when OP is a manipulation operation.

Lastly, if OP is a query operation rather than a manipulation operation, then the transformation from D_2 back to D_1 using g is no longer necessary. This is the only change needed in ALG_1; in the proof above, we may use $\mathit{time}(g, ALG_2(f(x_1))) = 0$. The requirement that $\mathit{time}(g, ALG_2(f(x_1))) \leq p(\mathit{time}(ALG_2, x_2))$ now holds vacuously. □

G.3 Rapidity Relations between Data Structures

Here we prove the rapidity relations between data structures studied in the paper as stated in Theorem 5.6, restated below with proof.

Theorem 5.6. The rapidity relations in Figure 5.4 hold.

Proof. The relation between QMDD and MPS is proved in Theorem 5.7 as restated in Section G.4. Finally, Section G.5 provides the transformations between QDDs that fulfill the conditions of Theorem 5.5. □

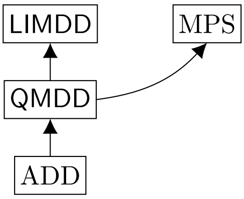


Figure G.1: Rapidity relations between data structures considered here. A solid arrow $D_1 \to D_2$ means D_2 is at least as rapid as D_1 for all operations satisfying A1 and A2 of Theorem 5.5.

G.4 MPS is at least as Rapid as QMDD

This appendix proves Theorem 5.7 from Sec. 5.5.2 by providing transformations between MPS and QMDD that realize the sufficient conditions of Theorem 5.5. The introduction to QDDs, given in Section 2.3, is relevant here.

Theorem 5.7. MPS is at least as rapid as QMDD for all operations satisfying A1 and A2.

Proof. Let f be the polynomial-time transformation from Lemma G.2. Let g be the weakly minimizing transformation from MPS to QMDD of Lemma G.3, that runs in time polynomial in the size of the input MPS and the resulting QMDD. These transitions satisfy requirements A3 and A4 of Theorem 5.5 respectively. Since QDDs are canonical data structures as explained in Section 5.2, all algorithms are by definition runtime monotonic, as for any state $|\varphi\rangle$ there is only one structure representing it, i.e., D^{φ} is a singleton set. This satisfies A2. Since its premise fulfills A1, the theorem follows. $\square$

Lemma G.2 (QMDD to MPS)**.** In polynomial time, a QMDD can be converted to an MPS representing the same state.

Proof. Consider a QMDD with root edge $\xrightarrow{\lambda}\!\textcircled{v}$ describing a state $|\varphi\rangle = \sum_{\vec{x}\in\{0,1\}^n} \alpha(\vec{x})\,|\vec{x}\rangle$. We will construct an MPS A describing the same state. For the purposes of this proof, we will call low edges *0-edges* and high edge *1-edges*.

First, without loss of generality, we may assume that the root edge label is $\lambda = 1$. Namely, we may multiply the labels on the root's low and high edges with λ, and then set the root edge label to 1; this operation preserves the state represented by the QMDD.

Denote by D_ℓ the number of nodes at the ℓ-th layer in QMDD v, i.e. $D_n = 1$ (the root node v) and $D_0 = 1$ (the leaf $\boxed{1}$). Recall that the QMDD is a directed, weighted graph whose vertices are divided into $n+1$ layers, i.e., the edges only connect nodes from consecutive layers. Therefore, we may speak of the $D_\ell \times D_{\ell-1}$ bipartite adjacency matrix between layer ℓ and layer $\ell-1$ of the diagram. For layer $1 \leq \ell \leq n$ and $x = 0, 1$, let A_ℓ^x be the $D_\ell \times D_{\ell-1}$ bipartite adjacency matrix obtained in this way using only the low edges if $x = 0$, and only the high edges if $x = 1$. That is, assuming some order on nodes within each level, the entry of the matrix A_ℓ^x in row r and column c is defined as

$$(A_\ell^x)_{r,c} = \begin{cases} \text{label}(e) & \text{if node with index } r \text{ in level } \ell \text{ has a } x\text{-edge } e \\ & \text{to node with index } c \text{ in level } \ell - 1 \\ & \\ 0 & \text{otherwise} \end{cases} \tag{G.2}$$

We claim that the following MPS A describes the same state as the QMDD:

$$A = (A_1^0, A_1^1, \ldots, A_n^0, A_n^1) \tag{G.3}$$

Following the MPS definition in Section 5.2, our claim is proven by showing that for QMDD root node v representing $|v\rangle$, we have

$$\langle \vec{x} | v \rangle = A_n^{x_n} \cdot A_{n-1}^{x_{n-1}} \cdots A_1^{x_1} \qquad \text{for all } \vec{x} \in \{0,1\}^n \tag{G.4}$$

For an n-qubit QMDD v, the amplitude $\langle \vec{x} | v \rangle$ for $\vec{x} \in \{0,1\}^n$ is equal to the product of the weights found on the single path from the root node node to leaf effected by $\vec{x}$ (this path is found as follows: go down from root to leaf; at a vertex at layer j, choose to traverse the low edge if $x_j = 0$ and the high edge if $x_j = 1$). We next reason that this product equals the single entry of the product $y := A_n^{x_n} \cdot A_{n-1}^{x_{n-1}} \cdots A_1^{x_1}$ from Equation G.4.

We recall several useful facts from graph theory. If G (G') is a weighted, directed bipartite graph on the bipartition $M \cup M''$ ($M'' \cup M'$) vertices, with weighted adjacency matrix A_G ($A_{G'}$), then it is not hard to see that the element $(A_G \cdot A_{G'})_{r,c}$ is the sum, over all two-step paths $r - a - c$ starting at vertex $r \in M$ and going through vertex $a \in M''$ to vertex $c \in M'$, of products of the two weights $w_{r \to a}$ and $w_{a \to c}$. More generally, for a sequence of weighted, directed bipartite graphs G_j with vertex set $M_j \cup M_{j+1}$, the (r, c)-th entry of the product of adjacency matrices $A_{G_1} \cdot A_{G_2} \cdot \ldots \cdot A_{G_n}$ equals $\sum_{\text{paths } \pi \text{ from } r \text{ to } c} \prod_{\text{edge} \epsilon \in \pi} \text{weight}(\epsilon)$.

Now note that the matrix y has dimensions 1×1 (since $D_0 = D_n = 1$), corresponding to a single root and single leaf. By the reasoning above, since y is the product of all bipartite adjacency matrices of the QMDD, the single element of this matrix is equal to the product of weights found on the single path from root to leaf as represented by $\vec{x}$. □

Lemma G.3 (MPS to QMDD)**.** There is a weakly minimizing transformation from MPS to QMDD, that runs in time polynomial in the size of the input MPS and the resulting QMDD.

Proof. Algorithm 23 shows the algorithm which converts an MPS to a QMDD. The idea is to use perform backtracking to construct the QMDD bottom-up. Specifically, given an MPS $\{A_n^0, A_n^1, \ldots, A_1^0, A_1^1\}$ representing a state $|\varphi\rangle = |0\rangle |\varphi_0\rangle + |1\rangle |\varphi_1\rangle$, the MPS for $|\varphi_0\rangle$ is easily constructed by setting the first open index to 0 and contracting these two blocks, i.e., $A_{n-1}^0 := A_n^0 \cdot A_{n-1}^0$ and $A_{n-1}^1 := A_n^0 \cdot A_{n-1}^1$, and similarly for $|\varphi_1\rangle$. We then recurse, constructing MPS for states $|\varphi_{00}\rangle, |\varphi_{01}\rangle$, etc. When we find a state whose QMDD node we have already constructed, then we may simply return an edge to that QMDD node without recursing further. This dynamic programming behavior is implemented through the check at Line 3.

Through the use of dynamic programming with the cache set D, it is clear that the number of recursive calls to MPS2QMDD is bound by the number of edges in the resulting QMDD. Dynamic programming is implemented by checking, for each call with MPS M, whether some QMDD node $v \in D$ already represents $|M\rangle$ up to a complex factor. To this end, the subroutine Equivalent, on Line 3, checks whether $|M\rangle = \lambda \cdot |v\rangle$ for some $\lambda \in \mathbb{C}$. It is straightforward to see that it runs in polynomial time in the sizes of QMDD v and MPS M: first, it creates an MPS for the given QMDD node v using the efficient transformation in Section G.4. Next, it computes several inner products on MPS, which can also be done in polynomial time, using the results in App. F. This Equivalent operation is called $|D|$ time, which dominates the runtime of each call MPS2QMDD. Therefore the entire runtime is polynomial in the sizes of the MPS and the resulting QMDD.

Since QMDD is canonical, the transformation is weakly minimizing by definition. ☐

Algorithm 23 An algorithm which converts an MPS into a QMDD. It runs in time polynomial in $s + d$, where s is the size of the QMDD, and d is the bond dimension of the MPS. Here D is the diagram representing the state. The subroutine Equivalent(v, M) computes whether the vectors $|v\rangle, |M\rangle$ are co-linear, i.e., whether there exists $\lambda \in \mathbb{C}$ such that $|M\rangle = \lambda |v\rangle$.

```
$D := \{\boxed{1}\}$    ▷ Initiate diagram $D$ with only a QMDD leaf node representing 1
procedure MPS2QMDD(MPS $M = \{A_j^x\}$)    ▷ Returns a root edge $e_R$ such that $|e_R\rangle = |M\rangle$
    if $D$ contains a node $v$ with $|M\rangle = \lambda |v\rangle$ then return $\xrightarrow{\lambda}\textcircled{v}$    ▷ Implemented with Equivalent$(v, M)$ for all $v \in D$
    Edge $e_0$ := MPS2QMDD($\{A_n^0 \cdot A_{n-1}^0, A_n^0 \cdot A_{n-1}^1\} \cup \{A_{n-2}^0, A_{n-2}^1, \ldots, A_1^0, A_1^1\}$)
    Edge $e_1$ := MPS2QMDD($\{A_n^1 \cdot A_{n-1}^0, A_n^1 \cdot A_{n-1}^1\} \cup \{A_{n-2}^0, A_{n-2}^1, \ldots, A_1^0, A_1^1\}$)
    Node $w := \bigcirc \overset{e_0}{\cdots\cdots} \bigcirc \overset{e_1}{\text{———}} \bigcirc$    ▷ Create new node $w$ with MakeNode
    $D := D \cup \{w\}$
    return Edge $\xrightarrow{1}\textcircled{w}$
procedure Equivalent( QMDD Node $v$, MPS $M = \{A_j^x\}$)
    $V :=$ QMDD2MPS$(v)$    ▷ Using transformation in Section G.4
    $s_V := \sqrt{|\langle V|V\rangle|}$    ▷ Compute inner product
    $s_M := \sqrt{|\langle M|M\rangle|}$    ▷ Compute inner product
    $\lambda := {}^1/_{s_V \cdot s_M} \langle V|M\rangle$    ▷ Compute inner product
    if $|\lambda| = 1$ then return "$|M\rangle = \frac{s_M}{s_V}\lambda |v\rangle$"
    else return "$|v\rangle$ is not equivalent to $|M\rangle$"
```

G.5 Transformations between QDDs

QDDs are canonical data structures as explained in Section 5.2 and Chapter 2. Therefore, (i) all algorithms are by definition runtime monotonic, as for any state $|\varphi\rangle$ there is only one structure representing it, i.e., D^φ is a singleton set; and (ii) all transformations given below are therefore weakly minimizing since they convert to a canonical data structure (namely, since they map to the unique element in D^φ, in particular they map to the minimum-size element of D^φ).

Algorithm 24 An algorithm which converts a LIMDD into an QMDD.

```
QMDD $D := \{ \xrightarrow{1}\textcircled{1} \}$                ▷ The QMDD is initialized to contain only the Leaf
procedure LIMDD 2QMDD(LIMDD edge $\xrightarrow{\lambda P_n \otimes P'}\textcircled{v}$)
Returns (a pointer to) an edge to a QMDD node
    if $v$ is the Leaf node then return $\xrightarrow{\lambda}\textcircled{v}$
    $R :=$ getLexminLabel($P_n \otimes P', v$)
    if the Cache contains tuple $(R, v)$ then return $\lambda \cdot$ Cache$[R, v]$
    for $x = 0, 1$ do
        QMDD edge $r_x :=$ Follow$_x$($\xrightarrow{P_n \otimes P'}\textcircled{v}$)
    QMDD edge $r :=$ MakeEdge($r_0, r_1$)
    Cache$[R, v] := r$
    $D := D \cup \{r\}$                ▷ Add the new edge to the diagram
    return $\lambda \cdot r$
```

G.5.1 Transforming LIMDD to QMDD

Algorithm 24 converts a LIMDD to a QMDD in time linear in the size of the output. The diagram is the set of edges D, which is initialized to contain the Leaf (i.e., the node $\xrightarrow{1}\textcircled{1}$), and is filled with the other edges during the recursive calls to LIMDD 2QMDD. The function getLexminLabel is taken from Vinkhuijzen et al. [337]; it returns a canonical edge label.

G.5.2 Transforming QMDD to LIMDD

By definition, a QMDD can be seen as a LIMDD in which every edge is labeled with a complex number and the n-qubit identity tensor $\mathbb{I}^{\otimes n}$. Thus, a transformation does not need to do anything. Optionally, it is possible to convert a given LIMDD to one of minimum size, as described by [337].

G.5.3 Transforming ADD to QMDD

To convert an ADD into a QMDD, we add a Leaf node labelled with 1; then, for each Leaf node labelled with $\lambda \neq 1$, we label each incoming edge with λ, and then reroute this edge to the (new) Leaf node labelled with 1. Optionally, the resulting QMDD can be minimized to obtain the canonical instance for this state, using, e.g., techniques from [59, 227].

G.5.4 Transforming QMDD to ADD

Algorithm 25 gives a method which converts an QMDD to an ADD. It is very similar to the ones used in the transformation LIMDD to QMDD above, . We here check whether the diagram already contains a function which is pointwise equal to the one we are currently considering. If so, we reuse that node; otherwise, we recurse.

Algorithm 25 An algorithm which converts an QMDD to an ADD. Its input is an QMDD edge e representing a state $|e\rangle$ on n qubits. Here the method $\textsc{Follow}_x(e)$ returns an QMDD edge representing the state $\langle x| \otimes \mathbb{I}^{\otimes n-1} \cdot |e\rangle$. It outputs an QMDD node w representing $|w\rangle = |e\rangle$.

```
procedure QMDD 2ADD(QMDD edge $e = \overset{\lambda}{\longrightarrow}\textcircled{v}$ on $n$ qubits)
    if $n = 0$ then
        $w := \longrightarrow\textcircled{\lambda}$
    else if $A$ contains a node $w$ with $|v\rangle = |w\rangle$ then
        return $w$
    else
        for $x = 0, 1$ do
            $w_x :=$ SLDD2ADD($\textsc{Follow}_x(e)$)
        QMDD Node $w := \textcircled{w_0}\cdots\bigcirc\!-\!\textcircled{w_1}$
    $A := A \cup \{w\}$
    return $w$
```

Bibliography

[1] Scott Aaronson. Multilinear formulas and skepticism of quantum computing. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 118–127, New York, NY, USA, 2004. Association for Computing Machinery.

[2] Scott Aaronson. *Quantum computing since Democritus.* Cambridge University Press, 2013.

[3] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5), nov 2004.

[4] Afshin Abdollahi and Massoud Pedram. Analysis and synthesis of quantum circuits by using quantum decision diagrams. In *Design, Automation and Test in Europe*, pages 317–322, 2006.

[5] Farid Ablayev, Aida Gainutdinova, and Marek Karpinski. On computational power of quantum branching programs. In *Fundamentals of Computation Theory: 13th International Symposium, FCT 2001 Riga, Latvia, August 22–24, 2001 Proceedings 13*, pages 59–70. Springer, 2001.

[6] Smaran Adarsh, Lukas Burgholzer, Tanmay Manjunath, and Robert Wille. SyReC synthesizer: An MQT tool for synthesis of reversible circuits. *Software Impacts*, 14:100451, 2022.

[7] Sheldon B. Akers. Binary decision diagrams. *IEEE Computer Architecture Letters*, 27(06):509–516, 1978.

[8] Anas N. Al-Rabadi, Marek Perkowski, and Martin Zwick. A comparison of modified reconstructability analysis and Ashenhurst-Curtis decomposition of Boolean functions. *Kybernetes*, 2004.

[9] F Aloul, I Markov, and K Sakallah. Mince: A static global variable-ordering for SAT and BDD. In *International Workshop on Logic and Synthesis*, pages 1167–1172, 2001.

[10] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. Association for Computing Machinery.

[11] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.

[12] Andris Ambainis, András Gilyén, Stacey Jeffery, and Martins Kokainis. Quadratic speedup for finding marked vertices by quantum walks. In *Symp. on Theory of Computing*, pages 412–424, 2020.

[13] Matthew Amy. *Formal methods in quantum circuit design.* PhD thesis, 2019.

[14] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time T-depth optimization of Clifford+ T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014.

[15] Matthew Amy, Martin Roetteler, and Krysta M Svore. Verified compilation of space-efficient reversible circuits. In *International Conference on Computer Aided Verification*, pages 3–21. Springer, 2017.

[16] Linda Anticoli, Carla Piazza, Leonardo Taglialegne, and Paolo Zuliani. Towards quantum programs verification: from quipper circuits to QPMC. In *Reversible Computation: 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings 8*, pages 213–219. Springer, 2016.

[17] Ebrahim Ardeshir-Larijani, Simon J Gay, and Rajagopal Nagarajan. Equivalence checking of quantum protocols. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*, pages 478–492. Springer, 2013.

[18] Ebrahim Ardeshir-Larijani, Simon J Gay, and Rajagopal Nagarajan. Verification of concurrent quantum protocols by equivalence checking. In *TACAS*, pages 500–514. Springer, 2014.

[19] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[20] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[21] Srinivasan Arunachalam, Sergey Bravyi, Chinmay Nirkhe, and Bryan O'Gorman. The parameterized complexity of quantum verification. *arXiv preprint arXiv:2202.08119*, 2022.

[22] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of an International Symposium on the theory of Switching, April 1957*, 1957.

[23] Gilles Audemard, Frédéric Koriche, and Pierre Marquis. On tractable XAI queries based on compiled representations. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 17, pages 838–849, 2020.

[24] Koenraad M R Audenaert and Martin B Plenio. Entanglement on mixed stabilizer states: normal forms and reduction procedures. *New Journal of Physics*, 7(1):170, 2005.

[25] UCLA Automated Reasoning Group. The SDD package. `http://reasoning.cs.ucla.edu/sdd/`, 2018.

[26] Junaid Babar, Chuan Jiang, Gianfranco Ciardo, and Andrew Miner. Binary decision diagrams with edge-specified reductions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–318. Springer, 2019.

[27] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 188–191, 1993.

[28] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[29] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[30] Pedro Baltazar, Rohit Chadha, and Paulo Mateus. Quantum computation tree logic—model checking and complete calculus. *International Journal of Quantum Information*, 6(02):219–236, 2008.

[31] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *ATVA 2017*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.

[32] Andreas Bärtschi and Stephan Eidenbenz. Deterministic preparation of Dicke states. In *Fundamentals of Computation Theory: 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12-14, 2019, Proceedings 22*, pages 126–139. Springer, 2019.

[33] Jon Barwise. *Handbook of mathematical logic*. Elsevier, 1982.

[34] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. symQV: Automated symbolic verification of quantum programs. In *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*, pages 181–198. Springer, 2023.

[35] Stephane Beauregard. Circuit for Shor's algorithm using $2n + 3$ qubits. *arXiv preprint quant-ph/0205095*, 2002.

[36] John S Bell. On the Einstein Podolsky Rosen paradox. *Physics Physique Fizika*, 1(3):195, 1964.

[37] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4(4):043001, nov 2019.

[38] Charles H. Bennet. Quantum cryptography: public key distribution and coin tossing. In *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore, Dec. 1984*, pages 175–179, 1984.

[39] Charles H. Bennett, Herbert J. Bernstein, Sandu Popescu, and Benjamin Schumacher. Concentrating partial entanglement by local operations. *Physical Review A*, 53(4):2046, 1996.

[40] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters*, 70(13):1895, 1993.

[41] Charles H. Bennett, David P. DiVincenzo, John A. Smolin, and William K Wootters. Mixed-state entanglement and quantum error correction. *Physical Review A*, 54(5):3824, 1996.

[42] Charles H. Bennett and Stephen J. Wiesner. Communication via one-and two-particle operators on Einstein-Podolsky-Rosen states. *Physical Review Letters*, 69(20):2881, 1992.

[43] Lucas Berent, Lukas Burgholzer, and Robert Wille. Towards a SAT encoding for quantum circuits: A journey from classical circuits to Clifford circuits and beyond. *arXiv:2203.00698*, 2022.

[44] David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1):47–66, 2016.

[45] Valeria Bertacco. The disjunctive decomposition of logic functions. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'97), November 1997*, pages 78–82, 1997.

[46] Valeria Bertacco and Maurizio Damiani. Boolean function representation based on disjoint-support decompositions. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 27–32. IEEE, 1996.

[47] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[48] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S Kottmann, Tim Menke, et al. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1):015004, 2022.

[49] Debjyoti Bhattacharjee and Anupam Chattopadhyay. Depth-optimal quantum circuit placement for arbitrary topologies. *arXiv preprint arXiv:1703.08540*, 2017.

[50] Jean-François Biasse and Fang Song. Efficient quantum algorithms for computing class groups and solving the principal ideal problem in arbitrary degree number fields. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 893–902. SIAM, 2016.

[51] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 193–207. Springer, 1999.

[52] Antonio Blanca, Zongchen Chen, Daniel Štefankovič, and Eric Vigoda. Hardness of identity testing for restricted Boltzmann machines and Potts models. *The Journal of Machine Learning Research*, 22(1):6727–6782, 2021.

[53] Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

[54] Beate Bollig and Matthias Buttkus. On the relative succinctness of sentential decision diagrams. *Theory of Computing Systems*, 63(6):1250–1277, 2019.

[55] Beate Bollig and Martin Farenholtz. On the relation between structured d-DNNFs and SDDs. *Theory of Computing Systems*, 65(2):274–295, 2021.

[56] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[57] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. On the complexity of quantum circuit compilation. In *Proceedings of the International Symposium on Combinatorial Search*, volume 9, pages 138–142, 2018.

[58] Simone Bova. SDDs are exponentially more succinct than OBDDs. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[59] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE, 1990.

[60] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, Texas, USA, January 23-25, 2011. Proceedings 12*, pages 70–87. Springer, 2011.

[61] Sergey Bravyi, Dan Browne, Padraic Calpin, Earl Campbell, David Gosset, and Mark Howard. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum*, 3:181, September 2019.

[62] Sergey Bravyi and David Gosset. Improved classical simulation of quantum circuits dominated by clifford gates. *Physical Review Letters*, 116:250501, Jun 2016.

[63] Sergey Bravyi and Alexei Kitaev. Universal quantum computation with ideal clifford gates and noisy ancillas. *Physical Review A*, 71:022316, Feb 2005.

[64] Sergey Bravyi, Graeme Smith, and John A. Smolin. Trading classical and quantum computational resources. *Physical Review X*, 6:021043, Jun 2016.

[65] Hans J. Briegel and Robert Raussendorf. Persistent entanglement in arrays of interacting particles. *Physical Review Letters*, 86:910–913, Jan 2001.

[66] Dan Browne and Hans Briegel. One-way quantum computation. *Quantum information: From foundations to quantum technology applications*, pages 449–473, 2016.

[67] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[68] Randal E Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–98. Springer, 2018.

[69] Yirng-An Chen Randal E Bryant. Verification of arithmetic circuits with binary moment diagrams. In *32nd Design Automation Conference*, pages 535–541. IEEE, 1995.

[70] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[71] Lukas Burgholzer, Richard Kueng, and Robert Wille. Random stimuli generation for the verification of quantum circuits. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 767–772, 2021.

[72] Lukas Burgholzer, Alexander Ploier, and Robert Wille. Tensor networks or decision diagrams? Guidelines for classical quantum circuit simulation. *arXiv preprint arXiv:2302.06616*, 2023.

[73] Lukas Burgholzer, Rudy Raymond, and Robert Wille. Verifying results of the IBM Qiskit quantum circuit compilation flow. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 356–365. IEEE, 2020.

[74] Lukas Burgholzer and Robert Wille. Improved DD-based equivalence checking of quantum circuits. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132. IEEE, 2020.

[75] Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 40(9):1810–1824, 2021.

[76] Padraic Calpin. *Exploring Quantum Computation Through the Lens of Classical Simulation*. PhD thesis, UCL (University College London), 2020.

[77] Jacques Carette, Gerardo Ortiz, and Amr Sabry. Symbolic execution of Hadamard-Toffoli quantum circuits. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, pages 14–26, 2023.

[78] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017.

[79] Rohit Chadha, Paulo Mateus, and Amílcar Sernadas. Reasoning about imperative quantum programs. *Electronic Notes in Theoretical Computer Science*, 158:19–39, 2006.

[80] Jing Chen, Song Cheng, Haidong Xie, Lei Wang, and Tao Xiang. Equivalence of restricted Boltzmann machines and tensor network states. *Physical Review B*, 97(8):085104, 2018.

[81] Tian-Fu Chen, Jie-Hong R Jiang, and Min-Hsiu Hsieh. Partial equivalence checking of quantum circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 594–604. IEEE, 2022.

[82] 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(1):303–326, 2014.

[83] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *27th AAAI Conference on Artificial Intelligence*, 2013.

[84] E. M. Clarke, K. L. McMillan, X Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 54–60, New York, NY, USA, 1993. Association for Computing Machinery.

[85] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[86] Edmund M Clarke, M. Fujita, P C McGeer, K. McMillan, J C-Y Yang, and X Zhao. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. 2 2001.

[87] Edmund M Clarke, Kenneth L McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proceedings of the 30th international Design Automation Conference*, pages 54–60, 1993.

[88] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[89] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, 2011.

[90] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

[91] Giso H. Dal, Alfons W Laarman, Arjen Hommersom, and Peter JF Lucas. A compositional approach to probabilistic knowledge compilation. *International Journal of Approximate Reasoning*, 138:38–66, 2021.

[92] Maurizio Damiani and Valeria Bertacco. Finding complex disjunctive decompositions of logic functions. In *Proc of the International Workshop on Logic & Synthesis*, pages 478–483, 1998.

[93] Aidan Dang, Charles D. Hill, and Lloyd C. L. Hollenberg. Optimising Matrix Product State Simulations of Shor's Algorithm. *Quantum*, 3:116, January 2019.

[94] Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *IJCAI*, volume 99, pages 284–289. Citeseer, 1999.

[95] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647, 2001.

[96] Adnan Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, pages 819–826. AAAI Press, 2011.

[97] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[98] JW De Bakker and Lambert G. L. T. Meertens. On the completeness of the inductive assertion method. *Journal of Computer and System Sciences*, 11(3):323–357, 1975.

[99] Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. Compressing probabilistic prolog programs. *Machine learning*, 70:151–168, 2008.

[100] Ellie D'hondt and Prakash Panangaden. Quantum weakest preconditions. *Mathematical Structures in Computer Science*, 16(3):429–451, 2006.

[101] Robert H Dicke. Coherence in spontaneous radiation processes. *Physical Review*, 93(1):99, 1954.

[102] Edsger W Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[103] Rolf Drechsler, Andisheh Sarabi, Michael Theobald, Bernd Becker, and Marek A Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proceedings of the 31st annual Design Automation Conference*, pages 415–419, 1994.

[104] Vincent Dumoulin, Ian Goodfellow, Aaron Courville, and Yoshua Bengio. On the challenges of physical implementations of RBMs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1), Jun. 2014.

[105] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum*, 4:279, June 2020.

[106] Vedran Dunjko and Hans J Briegel. Machine learning & artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics*, 81(7):074001, 2018.

[107] Wolfgang Dür, Guifre Vidal, and J Ignacio Cirac. Three qubits can be entangled in two inequivalent ways. *Physical Review A*, 62(6):062314, 2000.

[108] Pavol Ďuriš, Juraj Hromkovič, Stasys Jukna, Martin Sauerhoff, and Georg Schnitger. On multi-partition communication complexity. *Information and computation*, 194(1):49–75, 2004.

[109] Matthias Englbrecht and Barbara Kraus. Symmetries and entanglement of stabilizer states. *Physical Review A*, 101:062302, Jun 2020.

[110] Glen Evenbly and Guifré Vidal. Tensor network states and geometry. *Journal of Statistical Physics*, 145:891–918, 2011.

[111] Liangda Fang, Biqing Fang, Hai Wan, Zeqi Zheng, Liang Chang, and Quan Yu. Tagged sentential decision diagrams: Combining standard and zero-suppressed compression and trimming rules. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[112] Hélene Fargier and Pierre Marquis. Extending the knowledge compilation map: Krom, horn, affine and beyond. In *AAAI*, pages 442–447, 2008.

[113] Hélene Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[114] Hélene Fargier, Pierre Marquis, and Nicolas Schmidt. Semiring labelled decision diagrams, revisited: Canonicity and spatial efficiency issues. In *IJCAI*, pages 884–890, 2013.

[115] David Y Feinstein and Mitchell A Thornton. On the skipped variables of quantum multiple-valued decision diagrams. In *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pages 164–169. IEEE, 2011.

[116] Yuan Feng, Ernst Moritz Hahn, Andrea Turrini, and Lijun Zhang. QPMC: A model checker for quantum programs and protocols. In *FM 2015: Formal Methods: 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings 20*, pages 265–272. Springer, 2015.

[117] Yuan Feng, Nengkun Yu, and Mingsheng Ying. Model checking quantum Markov chains. *Journal of Computer and System Sciences*, 79(7):1181–1198, 2013.

[118] Felipe Cavalcanti Ferreira. *An Exploratory Study on the Usage of Quantum Programming Languages*. PhD thesis, 2022.

[119] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7), 1982.

[120] Richard P. Feynman. Quantum mechanical computers. *Optics news*, 11(2):11–20, 1985.

[121] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[122] WMC Foulkes, Lubos Mitas, RJ Needs, and Guna Rajagopal. Quantum Monte Carlo simulations of solids. *Reviews of Modern Physics*, 73(1):33, 2001.

[123] Lars-Hendrik Frahm and Daniela Pfannkuche. Ultrafast ab initio quantum chemistry using matrix product states. *Journal of Chemical Theory and Computation*, 15(4):2154–2165, 2019.

[124] Masahiro Fujita, Patrick C. McGeer, and JC-Y Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[125] Hector J Garcia, Igor L Markov, and Andrew W Cross. Efficient inner-product algorithm for stabilizer states. *arXiv preprint arXiv:1210.6646*, 2012.

[126] Sunita Garhwal, Maryam Ghorani, and Amir Ahmad. Quantum programming language: A systematic review of research topic and top cited languages. *Archives of Computational Methods in Engineering*, 28:289–310, 2021.

[127] Simon J Gay and Rajagopal Nagarajan. Communicating quantum processes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*, pages 145–157, 2005.

[128] Simon J Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. QMC: A model checker for quantum systems: Tool paper. In *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*, pages 543–547. Springer, 2008.

[129] Jordan Gergov and Christoph Meinel. Efficient boolean manipulation with obdd's can be extended to fbdd's. *IEEE Transactions on Computers*, 43(10):1197–1209, 1994.

[130] Jordan Gergov and Christoph Meinel. Mod-2-OBDDs—a data structure that generalizes exor-sum-of-products and ordered binary decision diagrams. *Formal Methods in System Design*, 8:273–282, 1996.

[131] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. Quantum cryptography. *Reviews of Modern Physics*, 74(1):145, 2002.

[132] Ivan Glasser, Nicola Pancotti, Moritz August, Ivan D Rodriguez, and J Ignacio Cirac. Neural-network quantum states, string-bond states, and chiral topological states. *Physical Review X*, 8(1):011006, 2018.

[133] Ivan Glasser, Ryan Sweke, Nicola Pancotti, Jens Eisert, and Ignacio Cirac. Expressive power of tensor-network factorizations for probabilistic modeling. *Advances in Neural Information Processing Systems*, 32, 2019.

[134] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification: 2nd International Conference, CAV'90 New Brunswick, NJ, USA, June 18–21, 1990 Proceedings 2*, pages 176–185. Springer, 1991.

[135] Daniel Gottesman. *Stabilizer codes and quantum error correction.* PhD thesis, California Institute of Technology, 1997.

[136] Daniel Gottesman. The Heisenberg representation of quantum computers. In *Proc. XXII International Colloquium on Group Theoretical Methods in Physics, 1998*, pages 32–43, 1998.

[137] Daniel Gottesman. Theory of fault-tolerant quantum computation. *Physical Review A*, 57:127–137, 1998.

[138] Daniel M Greenberger, Michael A Horne, Abner Shimony, and Anton Zeilinger. Bell's theorem without inequalities. *American Journal of Physics*, 58(12):1131–1143, 1990.

[139] David J Griffiths and Darrell F Schroeter. *Introduction to quantum mechanics.* Cambridge university press, 2018.

[140] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Symp. on Theory of Computing*, pages 212–219, 1996.

[141] Thomas Grurl, Jürgen Fuß, and Robert Wille. Considering decoherence errors in the simulation of quantum circuits using decision diagrams. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–7, 2020.

[142] Thomas Grurl, Jürgen Fuß, and Robert Wille. Lessons learnt in the implementation of quantum circuit simulation using decision diagrams. In *2021 IEEE 51st International Symposium on Multiple-Valued Logic (ISMVL)*, pages 87–92. IEEE, 2021.

[143] Thomas Grurl, Jürgen Fuß, and Robert Wille. Noise-aware quantum circuit simulation with decision diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[144] Thomas Grurl, Richard Kueng, Jürgen Fuß, and Robert Wille. Stochastic quantum circuit simulation using decision diagrams. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 194–199. IEEE, 2021.

[145] G. G. Guerreschi and A. Y. Matsuura. QAOA for max-cut requires hundreds of qubits for quantum speed-up. *Scientific Reports*, 9(1):6903, 2019.

[146] Wolfgang Günther and Rolf Drechsler. BDD minimization by linear transformations. In *In Advanced Computer Systems.* University Szczecin, 1998.

[147] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. IS-CAS MC: a web-based probabilistic model checker. In *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings 19*, pages 312–317. Springer, 2014.

[148] Sean Hallgren. Fast quantum algorithms for computing the unit group and class group of a number field. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 468–474, 2005.

[149] Sean Hallgren. Polynomial-time quantum algorithms for pell's equation and the principal ideal problem. *Journal of the ACM (JACM)*, 54(1):1–19, 2007.

[150] Thomas Häner and Damian S. Steiger. 5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2017.

[151] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using spin. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[152] Martin Hebenstreit, Richard Jozsa, Barbara Kraus, and Sergii Strelchuk. Computational power of matchgates with supplementary resources. *Physical Review A*, 102(5):052604, 2020.

[153] Marc Hein, Wolfgang Dür, Jens Eisert, Robert Raussendorf, M Nest, and H-J Briegel. Entanglement in graph states and its applications. In *Proceedings of the International School of Physics "Enrico Fermi"*, volume 162: Quantum Computers, Algorithms and Chaos. IOS Press, 2006.

[154] Marc Herbstritt. wld: A C++ library for decision diagrams. `https://ira.informatik.uni-freiburg.de/software/wld/`, 2004.

[155] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[156] Stefan Hillmich, Lukas Burgholzer, Florian Stögmüller, and Robert Wille. Reordering decision diagrams for quantum computing is harder than you might think. In *Reversible Computation: 14th International Conference, RC 2022, Urbino, Italy, July 5–6, 2022, Proceedings*, pages 93–107. Springer, 2022.

[157] Stefan Hillmich, Richard Kueng, Igor L. Markov, and Robert Wille. As accurate as needed, as efficient as possible: Approximations in DD-based quantum circuit simulation. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 188–193. IEEE, 2021.

[158] Stefan Hillmich, Igor L. Markov, and Robert Wille. Just like the real thing: Fast weak simulation of quantum computation. In *Design Automation Conference*, pages 1–6. IEEE, 2020.

[159] Gerard J. Holzmann. The model checker SPIN. *IEEE TSE*, 23:279–295, 1997.

[160] Gerard J. Holzmann, Eli Najm, and Ahmed Serhrouchni. SPIN model checking: An introduction. *International Journal on Software Tools for Technology Transfer*, 2:321–327, 2000.

[161] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 430–435, 2020.

[162] Xin Hong, Mingsheng Ying, Yuan Feng, Xiangzhen Zhou, and Sanjiang Li. Approximate equivalence checking of noisy quantum circuits. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 637–642, 2021.

[163] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. A tensor network based decision diagram for representation of quantum circuits. *ACM Trans. Des. Autom. Electron. Syst.*, 27(6), 2022.

[164] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[165] Yifei Huang and Peter Love. Approximate stabilizer rank and improved weak simulation of Clifford-dominated circuits for qudits. *Physical Review A*, 99:052307, May 2019.

[166] Yipeng Huang and Margaret Martonosi. QDB: from quantum algorithms towards correct quantum programs. *arXiv preprint arXiv:1811.05447*, 2018.

[167] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[168] Vít Jelínek. The rank-width of the square grid. *Discrete Applied Mathematics*, 158(7):841–850, 2010.

[169] Mark Jerrum and Kitty Meeks. The parameterised complexity of counting even and odd induced subgraphs. *Combinatorica*, 37(5):965–990, 2017.

[170] Wang Jian, Zhang Quan, and Tang Chao-Jing. Quantum secure communication scheme with W state. *Communications in Theoretical Physics*, 48(4):637, 2007.

[171] Phillip Johnston and Rozi Harris. The boeing 737 max saga: lessons for software organizations. *Software Quality Professional*, 21(3):4–12, 2019.

[172] N Cody Jones, James D Whitfield, Peter L McMahon, Man-Hong Yung, Rodney Van Meter, Alán Aspuru-Guzik, and Yoshihisa Yamamoto. Faster quantum chemistry simulation on fault-tolerant quantum computers. *New Journal of Physics*, 14(11):115023, 2012.

[173] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. Quest and high performance simulation of quantum computers. *Scientific reports*, 9(1):1–11, 2019.

[174] Bjarni Jónsson, Bela Bauer, and Giuseppe Carleo. Neural-network states for the classical simulation of quantum computing. *arXiv preprint arXiv:1808.05232*, 2018.

[175] Stephen Jordan. Quantum algorithm zoo. `https://quantumalgorithmzoo.org/`. Accessed: 20-05-2023.

[176] Richard Jozsa. Quantum algorithms and the Fourier transform. *Royal Society of London. Series A*, 454(1969):323–337, 1998.

[177] Richard Jozsa and Akimasa Miyake. Matchgates and classical simulation of quantum circuits. *Proceedings: Mathematical, Physical and Engineering Sciences*, pages 3089–3106, 2008.

[178] Gijs Kant et al. LTSmin: High-performance language-independent model checking. In *Tool and Algorithms for the Construction and Analysis of Systems (TACAS), TACAS'15*, volume 9035 of *LNCS*, pages 692–707. Springer, 2015.

[179] Pawel Kerntopf. A new heuristic algorithm for reversible logic synthesis. In *Proceedings of the 41st annual Design Automation Conference*, pages 834–837, 2004.

[180] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[181] Aleks Kissinger and John van de Wetering. Reducing T-count with the ZX-calculus. *arXiv preprint arXiv:1903.10477*, 2019.

[182] Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.

[183] Donald E Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[184] Lucas Kocia and Peter Love. Stationary phase method in discrete wigner functions and classical simulation of quantum circuits. *Quantum*, 5:494, 2021.

[185] Lucas Kocia and Mohan Sarovar. Improved simulation of quantum circuits by fewer gaussian eliminations. *arXiv:2003.01130*, 2020.

[186] Attila Kondacs and John Watrous. On the power of quantum finite state automata. In *Proceedings 38th annual symposium on foundations of computer science*, pages 66–75. IEEE, 1997.

[187] Fabrice Kordon, Hubert Garavel, Lom-Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, Francis Hulin-Hubard, Elvio Gilberto Amparore, Marco Beccuti, Bernard Berthomieu, Hugues Evrard, Peter Gjøl Jensen, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Jirí Srba, Yann Thierry-Mieg, Jaco van de Pol, and Karsten Wolf. Mcc'2017 - the seventh model checking contest. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, XIII:181–209, 2018.

[188] Fabrice Kordon, Hubert Garavel, Lom-Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, César Rodríguez, and Francis Hulin-Hubard. MCC'2015 - The Fifth Model Checking Contest. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, XI:262–273, 2016.

[189] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

[190] Richard Kueng and David Gross. Qubit stabilizer states are complex projective 3-designs. *arXiv preprint arXiv:1510.02767*, 2015.

[191] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 585–591. Springer, 2011.

[192] Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *IJCAI*, volume 17, pages 667–673, 2017.

[193] Y-T Lai, Massoud Pedram, and Sarma BK Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):959–975, 1994.

[194] Yong Lai, Dayou Liu, and Minghao Yin. New canonical representations by augmenting OBDDs with conjunctive decomposition. *Journal of Artificial Intelligence Research*, 58:453–521, 2017.

[195] Yong Lai, Kuldeep S. Meel, and Roland H.C. Yap. CCDD: A tractable representation for model counting and uniform sampling. *arXiv preprint arXiv:2202.10025*, 2022.

[196] Benjamin P Lanyon, James D Whitfield, Geoff G Gillett, Michael E Goggin, Marcelo P Almeida, Ivan Kassal, Jacob D Biamonte, Masoud Mohseni, Ben J Powell, Marco Barbieri, et al. Towards quantum chemistry on a quantum computer. *Nature Chemistry*, 2(2):106, 2010.

[197] Anna LD Latour, Behrouz Babaki, Anton Dries, Angelika Kimmig, Guy Van den Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and probabilistic programming: from knowledge compilation to constraint solving. In *Principles and Practice of Constraint Programming: 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 23*, pages 495–511. Springer, 2017.

[198] Anna Louise D Latour, Behrouz Babaki, and Siegfried Nijssen. Stochastic constraint propagation for mining probabilistic networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 1137–1145, 2019.

[199] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[200] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[201] Daniel A Lidar and Haobin Wang. Calculating the thermal rate constant with exponential speedup on a quantum computer. *Physical Review E*, 59(2):2429, 1999.

[202] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[203] Victoria Lipinska, Gláucia Murta, and Stephanie Wehner. Anonymous transmission in a noisy quantum network using the W state. *Physical Review A*, 98:052320, Nov 2018.

[204] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2):346–358, 1979.

[205] Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum Hoare logic. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 187–207. Springer, 2019.

[206] Wen Liu, Yong-Bin Wang, and Zheng-Tao Jiang. An efficient protocol for the quantum private comparison of equality with W state. *Optics Communications*, 284(12):3160–3163, 2011.

[207] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[208] Seth Lloyd. Universal quantum simulators. *Science*, 273(5278):1073–1078, 1996.

[209] Benjamin Lovitz and Vincent Steffan. New techniques for bounding stabilizer rank. *Quantum*, 6:692, 2022.

[210] LTSmin with sentential decision diagrams. `https://zenodo.org/record/3940936`.

[211] Chin-Yung Lu, Shiou-An Wang, and Sy-Yen Kuo. An extended XQDD representation for multiple-valued quantum logic. *IEEE Transactions on Computers*, 60(10):1377–1389, 2011.

[212] Eugene M Luks, Ferenc Rákóczi, and Charles RB Wright. Some algorithms for nilpotent permutation groups. *Journal of Symbolic Computation*, 23(4):335–354, 1997.

[213] Guanfeng Lv, Yao Chen, Yachao Feng, Qingliang Chen, and Kaile Su. A succinct and efficient implementation of a 2^{32} BDD package. In Tiziana Margaria, Zongyan Qiu, and Hongli Yang, editors, *Int'l Symp. on Theoretical Aspects of Software Engineering*, pages 241–244, 2012.

[214] James Martens, Arkadev Chattopadhya, Toni Pitassi, and Richard Zemel. On the representational efficiency of restricted Boltzmann machines. *Advances in Neural Information Processing Systems*, 26, 2013.

[215] Robert Mateescu, Rina Dechter, and Radu Marinescu. And/or multi-valued decision diagrams (AOMDDs) for graphical models. *Journal of Artificial Intelligence Research*, 33:465–519, 2008.

[216] Paulo Mateus, Jaime Ramos, Amílcar Sernadas, and Cristina Sernadas. Temporal logics for reasoning about quantum systems. *Semantic techniques in quantum computation*, pages 389–413, 2009.

[217] Paulo Mateus and Amílcar Sernadas. Weakly complete axiomatization of exogenous quantum propositional logic. *Information and Computation*, 204(5):771–794, 2006.

[218] Yusuke Matsunaga. An exact and efficient algorithm for disjunctive decomposition. *Proceedings of Synthesis and System Integration of Mixed Technologies (SASIMI'98, Japan), Oct.*, 1998.

[219] Sam McArdle, Suguru Endo, Alán Aspuru-Guzik, Simon C Benjamin, and Xiao Yuan. Quantum computational chemistry. *Reviews of Modern Physics*, 92(1):015003, 2020.

[220] James McClung. *Constructions and Applications of W-States.* PhD thesis, Worcester Polytechnic Institute, 2020.

[221] K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992. UMI No. GAX92-24209.

[222] Saeed Mehraban and Mehrdad Tahmasbi. Lower bounds on the approximate stabilizer rank: A probabilistic approach. *arXiv preprint arXiv:2305.10277*, 2023.

[223] Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. Read, write and copy dependencies for symbolic model checking. In *Haifa Verification Conference*, pages 204–219. Springer, 2014.

[224] Jeroen Meijer and Jaco van de Pol. Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In *NASA Formal Methods Symposium*, pages 255–271. Springer, 2016.

[225] Roger G Melko, Giuseppe Carleo, Juan Carrasquilla, and J Ignacio Cirac. Restricted Boltzmann machines in quantum physics. *Nature Physics*, 15(9):887–892, 2019.

[226] D Michael Miller, David Y Feinstein, and Mitchell A Thrornton. QMDD minimization using sifting for variable reordering. *Journal of Multiple-Valued Logic & Soft Computing*, 13, 2007.

[227] D Michael Miller and Mitchell A Thornton. QMDD: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, pages 30–30. IEEE, 2006.

[228] D Michael Miller, Robert Wille, and Zahra Sasanian. Elementary quantum gate realizations for multiple-control toffoli gates. In *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pages 288–293. IEEE, 2011.

[229] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th ACM/IEEE Design Automation Conference*, pages 272–277. IEEE, 1993.

[230] Shin-ichi Minato. Finding simple disjoint decompositions in frequent itemset data using zero-suppressed BDD. In *Proc. of IEEE ICDM 2005 workshop on Computational Intelligence in Data Mining*, pages 3–11, 2005.

[231] Ashley Montanaro. Quantum-walk speedup of backtracking algorithms. *Theory of Computing*, 14(1):1–24, 2018.

[232] Tomoyuki Morimae and Suguru Tamaki. Fine-grained quantum computational supremacy. *arXiv preprint arXiv:1901.01637*, 2019.

[233] MQT DDSIM - A quantum circuit simulator based on decision diagrams written in C++. `https://github.com/cda-tum/ddsim/tree/limdd`.

[234] Ece C Mutlu. Quantum probabilistic models using Feynman diagram rules for better understanding the information diffusion dynamics in online social networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 13730–13731, 2020.

[235] Kengo Nakamura, Shuhei Denzumi, and Masaaki Nishino. Variable shift SDD: a more succinct sentential decision diagram. *arXiv preprint arXiv:2004.02502*, 2020.

[236] Naoki Nakatani and Garnet Kin Chan. Efficient tree tensor network states (TTNS) for quantum chemistry: Generalizations of the density matrix renormalization group algorithm. *The Journal of chemical physics*, 138(13), 2013.

[237] Amit Narayan, Jawahar Jain, Masahiro Fujita, and Alberto Sangiovanni-Vincentelli. Partitioned ROBDDs-a compact, canonical and efficiently manipulable representation for boolean functions. In *Proceedings of International Conference on Computer Aided Design*, pages 547–554. IEEE, 1996.

[238] Hendrik Poulsen Nautrup, Nicolas Delfosse, Vedran Dunjko, Hans J Briegel, and Nicolai Friis. Optimizing quantum error correction codes with reinforcement learning. *Quantum*, 3:215, 2019.

[239] Xiaotong Ni, Oliver Buerschaper, and Maarten Van den Nest. A non-commuting stabilizer formalism. *Journal of Mathematical Physics*, 56(5):052201, 2015.

[240] Michael A Nielsen and Isaac L Chuang. Quantum information and quantum computation. *Cambridge: Cambridge University Press*, 2(8):23, 2000.

[241] Philipp Niemann, Robert Wille, and Rolf Drechsler. Equivalence checking in multi-level quantum systems. In *Reversible Computation: 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings 6*, pages 201–215. Springer, 2014.

[242] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. QMDDs: Efficient quantum function representation and manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):86–99, 2015.

[243] Philipp Niemann, Alwin Zulehner, Rolf Drechsler, and Robert Wille. Overcoming the trade-off between accuracy and compactness in decision diagrams for quantum computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[244] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer, 2002.

[245] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[246] Weber Noah, Niranjan Balasubramanian, and Nathanael Chambers. Event representations with tensor-based compositions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[247] Yusuke Nomura, Andrew S Darmawan, Youhei Yamaji, and Masatoshi Imada. Restricted boltzmann machine learning for solving strongly correlated quantum systems. *Physical Review B*, 96(20):205152, 2017.

[248] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[249] Umut Oztok and Adnan Darwiche. CV-width: A new complexity parameter for CNFs. In *ECAI*, pages 675–680, 2014.

[250] Sebastian Paeckel, Thomas Köhler, Andreas Swoboda, Salvatore R Manmana, Ulrich Schollwöck, and Claudius Hubig. Time-evolution methods for matrix-product states. *Annals of Physics*, 411:167998, 2019.

[251] Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: an empirical study. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[252] Christos H Papadimitriou et al. Computational complexity. 1994.

[253] Lawrence C Paulson. *Isabelle: A generic theorem prover.* Springer, 1994.

[254] Lawrence C Paulson. Isabelle: The next 700 theorem provers. *arXiv preprint cs/9301106*, 2000.

[255] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. Leveraging secondary storage to simulate deep 54-qubit sycamore circuits, 2019.

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

[257] Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking paradigms in quantum circuit design: A case study. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 517–522, 2022.

[258] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[259] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8:39–64, 1996.

[260] Shir Peleg, Amir Shpilka, and Ben Lee Volk. Lower bounds on stabilizer rank. *Quantum*, 6:652, 2022.

[261] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of Shor's factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023.

[262] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac. Matrix product state representations. *Quantum Information & Computation*, 7(5):401–430, jul 2007.

[263] Stefano Pirandola, Ulrik L Andersen, Leonardo Banchi, Mario Berta, Darius Bunandar, Roger Colbeck, Dirk Englund, Tobias Gehring, Cosmo Lupo, Carlo Ottaviani, et al. Advances in quantum cryptography. *Advances in Optics and Photonics*, 12(4):1012–1236, 2020.

[264] Stephen Plaza and Valeria Bertacco. Boolean operations on decomposed functions. *Proceedings of the 24th International Workshop on Logic & Synthesis*, pages 310–317, 2005.

[265] Stephen Plaza and Valeria Bertacco. STACCATO: disjoint support decompositions from BDDs through symbolic kernels. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 276–279, 2005.

[266] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[267] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[268] Hammam Qassim, Hakop Pashayan, and David Gosset. Improved upper bounds on the stabilizer rank of magic states. *Quantum*, 5:606, 2021.

[269] Mohamed Raed El Aoun, Heng Li, Foutse Khomh, and Lionel Tidjon. Bug characteristics in quantum software ecosystem. *arXiv preprint arXiv:2204.11965*, 2022.

[270] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. *arXiv preprint arXiv:1803.00699*, 2018.

[271] Robert Raussendorf and Hans J. Briegel. A one-way quantum computer. *Physical Review Letters*, 86:5188–5191, May 2001.

[272] Robert Raussendorf, Daniel E Browne, and Hans J Briegel. Measurement-based quantum computation on cluster states. *Physical Review A*, 68(2):022312, 2003.

[273] Igor Razgon. On OBDDs for CNFs of bounded treewidth. *arXiv preprint arXiv:1308.3829*, 2013.

[274] Markus Reiher, Nathan Wiebe, Krysta M Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences*, 114(29):7555–7560, 2017.

[275] José Ignacio Requeno and José Manuel Colom. Compact representation of biological sequences using set decision diagrams. In *6th International Conference on Practical Applications of Computational Biology & Bioinformatics*, pages 231–239. Springer, 2012.

[276] Michael Rice and Sanjay Kulhari. A survey of static variable ordering heuristics for efficient BDD/MDD construction. *University of California, Tech. Rep*, 2008.

[277] Leanna Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance.* CRC Press, 2017.

[278] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.

[279] Mehdi Saeedi and Igor L Markov. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys (CSUR)*, 45(2):1–34, 2013.

[280] Tuhin Sahai, Anurag Mishra, Jose Miguel Pasini, and Susmit Jha. Estimating the density of states of boolean satisfiability problems on classical and quantum computing platforms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1627–1635, Apr. 2020.

[281] Scott Sanner and David McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 1384–1390, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[282] Tsutomu Sasao. FPGA design by generalized functional decomposition. In *Logic synthesis and optimization*, pages 233–258. Springer, 1993.

[283] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *1998 International Workshop on Logic Synthesis*, pages 471–477, 1998.

[284] Ulrich Schollwöck. The density-matrix renormalization group in the age of matrix product states. *Annals of physics*, 326(1):96–192, 2011.

[285] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[286] Peter Selinger. Quantum circuits of T-depth one. *Physical Review A*, 87(4):042302, 2013.

[287] Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for quantum circuits as classical planning. *arXiv preprint arXiv:2304.12014*, 2023.

[288] Claude E Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.

[289] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. Knowledge compilation meets uniform sampling. In *LPAR*, pages 620–636, 2018.

[290] Y-Y Shi, L-M Duan, and Guifre Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical Review A*, 74(2):022320, 2006.

[291] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Jour. of Comp.*, 26(5):1484–1509, 1997.

[292] Radu I Siminiceanu and Gianfranco Ciardo. New metrics for static variable ordering in decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 90–104. Springer, 2006.

[293] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. CFLOBDDs: Context-free-language ordered binary decision diagrams. *arXiv preprint arXiv:2211.06818*, 2022.

[294] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. Symbolic quantum simulation with quasimodo. *arXiv preprint arXiv:2302.04349*, 2023.

[295] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. Symbolic quantum simulation with quasimodo. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 213–225, Cham, 2023. Springer Nature Switzerland.

[296] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. Weighted context-free-language ordered binary decision diagrams. *arXiv preprint arXiv:2305.13610*, 2023.

[297] SW Sloan. A FORTRAN program for profile and wavefront reduction. *International Journal for Numerical Methods in Engineering*, 28(11):2651–2679, 1989.

[298] Graeme Smith and Debbie Leung. Typical entanglement of stabilizer states. *Physical Review A*, 74(6):062314, 2006.

[299] Kaitlin N Smith and Mitchell A Thornton. A quantum computational compiler and design tool for technology-specific targets. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 579–588, 2019.

[300] Mathias Soeken, Stefan Frehse, Robert Wille, and Rolf Drechsler. RevKit: A toolkit for reversible circuit design. *J. Multiple Valued Log. Soft Comput.*, 18(1):55–65, 2012.

[301] Mathias Soeken, Laura Tague, Gerhard W. Dueck, and Rolf Drechsler. Ancilla-free synthesis of large reversible functions using binary decision diagrams. *Journal of Symbolic Computation*, 73:1–26, 2016.

[302] Mathias Soeken, Robert Wille, Christoph Hilken, Nils Przigoda, and Rolf Drechsler. Synthesis of reversible circuits with minimal lines for large functions. In *17th Asia and South Pacific Design Automation Conference*, pages 85–92. IEEE, 2012.

[303] Fabio Somenzi. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[304] Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0. `http://vlsi.colorado.edu/~fabio/`, 2015.

[305] Stabranksearcher: code for finding (upper bounds to) the stabilizer rank of a quantum state. `https://github.com/timcp/StabRankSearcher`, 2021.

[306] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[307] Ryan Sweke, Markus S Kesselring, Evert P.L. van Nieuwenburg, and Jens Eisert. Reinforcement learning decoders for fault-tolerant quantum computation. *Machine Learning: Science and Technology*, 2(2):025005, 2020.

[308] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2):243–270, 1997.

[309] Yasuhiro Takahashi and Noboru Kunihiro. A quantum circuit for Shor's factoring algorithm using $2n+2$ qubits. *Quantum Information & Computation*, 6(2):184–192, 2006.

[310] Barbara M Terhal. Quantum error correction for quantum memories. *Reviews of Modern Physics*, 87(2):307, 2015.

[311] Barbara M. Terhal and David P. DiVincenzo. Classical simulation of noninteracting-fermion quantum circuits. *Physical Review A*, 65:032325, Mar 2002.

[312] Dimitrios Thanos, Tim Coopmans, and Alfons Laarman. Fast equivalence checking of quantum circuits of clifford gates. *ATVA 2023 (accepted for publication)*, 2023.

[313] Himanshu Thapliyal, Edgard Munoz-Coreas, TSS Varun, and Travis S Humble. Quantum circuit designs of integer division optimizing T-count and T-depth. *IEEE Transactions on Emerging Topics in Computing*, 9(2):1045–1056, 2019.

[314] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H. Booth, et al. The variational quantum eigensolver: a review of methods and best practices. *Physics Reports*, 986:1–128, 2022.

[315] Giacomo Torlai, Guglielmo Mazzola, Juan Carrasquilla, Matthias Troyer, Roger Melko, and Giuseppe Carleo. Neural-network quantum state tomography. *Nature Physics*, 14(5):447–450, 2018.

[316] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. *Formal Methods in System Design*, 19:195–215, 2001.

[317] Andrea Turrini. An introduction to quantum model checking. *Applied Sciences*, 12(4):2016, 2022.

[318] Tomás E. Uribe and Mark E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In *International Conference on Constraints in Computational Logics*, pages 34–49. Springer, 1994.

[319] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990 10*, pages 491–515. Springer, 1991.

[320] John van de Wetering. ZX-calculus for the working quantum computer scientist. *arXiv preprint arXiv:2012.13966*, 2020.

[321] Ewout van den Berg and Kristan Temme. Circuit optimization of Hamiltonian simulation by simultaneous diagonalization of pauli clusters. *Quantum*, 4:322, 2020.

[322] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[323] M. Van den Nest, W. Dür, G. Vidal, and H. J. Briegel. Classical simulation versus universality in measurement-based quantum computation. *Physical Review A*, 75:012337, Jan 2007.

[324] Maarten Van den Nest, Jeroen Dehaene, and Bart De Moor. Graphical description of the action of local clifford transformations on graph states. *Physical Review A*, 69(2):022316, 2004.

[325] Maarten Van den Nest, Jeroen Dehaene, and Bart De Moor. Local unitary versus local clifford equivalence of stabilizer states. *Physical Review A*, 71:062323, Jun 2005.

[326] Tom Van Dijk, Alfons Laarman, and Jaco Van De Pol. Multi-core BDD operations for symbolic reachability. *Electronic Notes in Theoretical Computer Science*, 296:127–143, 2013.

[327] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-core decision diagrams. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 677–691. Springer, 2015.

[328] Tom van Dijk, Robert Wille, and Robert Meolic. Tagged BDDs: combining reduction rules from different decision diagram types. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 108–115. FMCAD Inc, 2017.

[329] Vivien Vandaele, Simon Martiel, Simon Perdrix, and Christophe Vuillot. Optimal Hadamard gate count for Clifford $+T$ synthesis of Pauli rotations sequences. *arXiv preprint arXiv:2302.07040*, 2023.

[330] Frank Verstraete, Diego Porras, and J. Ignacio Cirac. Density matrix renormalization group and periodic boundary conditions: A quantum information perspective. *Physical Review Letters*, 93(22):227205, 2004.

[331] George F. Viamontes, Igor L. Markov, and John P. Hayes. Improving gate-level simulation of quantum circuits. *Quantum Information Processing*, 2(5):347–380, 2003.

[332] George F Viamontes, Igor L Markov, and John P Hayes. *Quantum circuit simulation*. Springer Science & Business Media, 2009.

[333] George F. Viamontes, Manoj Rajagopalan, Igor L. Markov, and John P. Hayes. Gate-level simulation of quantum circuits. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 295–301, 2003.

[334] G.F. Viamontes, I.L. Markov, and J.P. Hayes. High-performance QuIDD-based simulation of quantum circuits. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1354–1355 Vol.2, 2004.

[335] Guifré Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 91(14):147902, 2003.

[336] Renaud Vilmart. Quantum multiple-valued decision diagrams in graphical calculi, 2021.

[337] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. LIMDD: A decision diagram for simulation of quantum computing including stabilizer states. *Quantum*, 7:1108, 2023.

[338] Lieuwe Vinkhuijzen, Thomas Grurl, Stefan Hillmich, Sebastiaan Brand, Robert Wille, and Alfons Laarman. Efficient implementation of LIMDDs for quantum circuit simulation. In *International Symposium on Model Checking Software*, pages 3–21. Springer, 2023.

[339] Lieuwe Vinkhuijzen and Alfons Laarman. Symbolic model checking with sentential decision diagrams. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 124–142. Springer, 2020.

[340] Hefeng Wang, Sabre Kais, Alán Aspuru-Guzik, and Mark R Hoffmann. Quantum algorithm for obtaining the energy spectrum of molecular systems. *Physical Chemistry Chemical Physics*, 10(35):5388–5393, 2008.

[341] Shiou-An Wang, Chin-Yung Lu, I-Ming Tsai, and Sy-Yen Kuo. An XQDD-based verification method for quantum circuits. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 91(2):584–594, 2008.

[342] Shiou-An Wang, Chin-Yung Lu, I-Ming Tsai, and Sy-Yen Kuo. An xqdd-based verification method for quantum circuits. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 91(2):584–594, 2008.

[343] Tianshu Wang, Yuexian Hou, Panpan Wang, and Xiaolei Niu. Exploring relevance judgement inspired by quantum weak measurement. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[344] Ingo Wegener. *Branching programs and binary decision diagrams: theory and applications.* SIAM, 2000.

[345] Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R Jiang. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 523–528, 2022.

[346] Steven R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69:2863–2866, Nov 1992.

[347] Robert Wille and Rolf Drechsler. BDD-based synthesis of reversible logic for large functions. In *Proceedings of the 46th Annual Design Automation Conference*, pages 270–275, 2009.

[348] Robert Wille and Rolf Drechsler. Effect of BDD optimization on synthesis of reversible and quantum logic. *Electronic Notes in Theoretical Computer Science*, 253(6):57–70, 2010.

[349] Robert Wille, Daniel Große, D Michael Miller, and Rolf Drechsler. Equivalence checking of reversible circuits. In *2009 39th International Symposium on Multiple-Valued Logic*, pages 324–330. IEEE, 2009.

[350] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. JKQ: JKU tools for quantum computing. In *Int'l Conf. on CAD*, pages 154:1–154:5, 2020.

[351] Robert Wille, Nils Przigoda, and Rolf Drechsler. A compact and efficient SAT encoding for quantum circuits. In *2013 Africon*, pages 1–6. IEEE, 2013.

[352] Nic Wilson. Decision diagrams for the computation of semiring valuations. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 331–336, 2005.

[353] Yu Wu, Per Austrin, Toniann Pitassi, and David Liu. Inapproximability of treewidth and related problems. *Journal of Artificial Intelligence Research*, 49:569–600, 2014.

[354] Yukai Wu, L.-M. Duan, and Dong-Ling Deng. Artificial neural network based computation for out-of-time-ordered correlators. *Physical Review B*, 101:214308, Jun 2020.

[355] Ming Xu, Jianling Fu, Jingyi Mei, and Yuxin Deng. Model checking QCTL plus on quantum markov chains. *Theoretical Computer Science*, 913:43–72, 2022.

[356] Shigeru Yamashita and Igor L Markov. Fast equivalence-checking for quantum circuits. In *2010 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 23–28. IEEE, 2010.

[357] Shigeru Yamashita, Shin-ichi Minato, and D Michael Miller. DDMF: An efficient decision diagram structure for design verification of quantum circuits under a practical restriction. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 91(12):3793–3802, 2008.

[358] Feidiao Yang, Jiaqing Jiang, Jialin Zhang, and Xiaoming Sun. Revisiting online quantum state learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):6607–6614, Apr. 2020.

[359] Mingsheng Ying. Floyd–Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):1–49, 2012.

[360] Mingsheng Ying and Yuan Feng. Model checking quantum systems—a survey. *arXiv preprint arXiv:1807.09466*, 2018.

[361] Mingsheng Ying and Yuan Feng. *Model Checking Quantum Systems: Principles and Algorithms*. Cambridge University Press, 2021.

[362] Mingsheng Ying, Yuan Feng, Runyao Duan, and Zhengfeng Ji. An algebra of quantum processes. *ACM Transactions on Computational Logic (TOCL)*, 10(3):1–36, 2009.

[363] Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. Model-checking linear-time properties of quantum systems. *ACM Transactions on Computational Logic (TOCL)*, 15(3):1–31, 2014.

[364] Christof Zalka. Simulating quantum systems on a quantum computer. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):313–322, 1998.

[365] Yuan-Hang Zhang, Zhian Jia, Yu-Chun Wu, and Guang-Can Guo. An efficient algorithmic way to construct Boltzmann machine representations for arbitrary stabilizer code. *arXiv:1809.08631*, 2018.

[366] Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. A quantum interpretation of bunched logic & quantum separation logic. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2021.

[367] Li Zhou, Nengkun Yu, and Mingsheng Ying. An applied quantum Hoare logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1149–1162, 2019.

[368] Alwin Zulehner, Stefan Hillmich, Igor L Markov, and Robert Wille. Approximation of quantum states using decision diagrams. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 121–126. IEEE, 2020.

[369] Alwin Zulehner, Stefan Hillmich, and Robert Wille. How to efficiently handle complex values? implementing decision diagrams for quantum computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019.

[370] Alwin Zulehner, Philipp Niemann, Rolf Drechsler, and Robert Wille. Accuracy and compactness in decision diagrams for quantum computation. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 280–283. IEEE, 2019.

[371] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2018.

[372] Alwin Zulehner and Robert Wille. Improving synthesis of reversible circuits: Exploiting redundancies in paths and nodes of QMDDs. In *International Conference on Reversible Computation*, pages 232–247. Springer, 2017.

[373] Alwin Zulehner and Robert Wille. One-pass design of reversible circuits: Combining embedding and synthesis for reversible logic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(5):996–1008, 2017.

[374] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):848–859, 2018.

[375] Alwin Zulehner and Robert Wille. Compiling $SU(4)$ quantum circuits to IBM QX architectures. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 185–190, 2019.

[376] Alwin Zulehner and Robert Wille. *Introducing design automation for quantum computing*, volume 11. Springer, 2020.

Summary

Quantum computers are a proposed fundamentally new type of computer. They aim to perform some computations much faster than previously possible by exploiting phenomena at the quantum scale, called superposition and entanglement. If and when large ones are be built, then they promise computational speedups for several important problems, notably in chemistry and physics, potentially contributing to important applications such as better medicines, batteries and solar panels, among others, thus contributing to and accelerating the transition to green energy and a sustainable economy. In order to realize these promises, we will need tools which analyze, simulate, compile, and verify these new computers and the algorithms that run on them. Meanwhile, of course, we still require tools to compile, analyze and verify classical software. The goal of this thesis is to contribute to a tool set for these two use cases.

Before we describe our contributions, however, let us describe the problem we are trying to solve and what makes it so difficult.

- When analyzing a quantum algorithm, we have a description of it in terms of a set of qubits and a series of quantum gates and measurements that are to be applied to the qubits. We wish to answer questions such as, “what is the most likely output?” or, “how likely is the computer to end up in a particular state?”

  The biggest difficulty is that, if a quantum computer consists of n qubits, then its state at any point in time is described by a list of 2^n complex numbers. This exponential increase makes it infeasible to store the list without using compression techniques when n becomes too large.

- In classical model checking, the goal is to decide whether an algorithm satisfies a specification. For example, we may wish to check that a program never enters

> a state where two threads receive access to a resource that is supposed to be mutually exclusive. The difficulty is that, if a system consists of n bits, then the set of possible states grows as 2^n – exponentially, again, in the number n of bits.

In both cases, a tool which analyzes the system will be successful if it manages to tackle the exponentially-sized state space. Because the challenges in the quantum and classical cases are so similar, this allows us to focus on tackling their shared challenge: to find enough structure in the state space to compress it into a manageable amount of space. A powerful tool may then deliver value for both the above use cases.

Our tool of choice in this thesis is the *decision diagram* (DD). Decision diagrams are a versatile data structure with a rich theory and with applications in many parts of computer science, including analyzing quantum computers and doing (classical) model checking. The role of decision diagrams in these scenarios is to provide a complete snapshot of the state of the computation at each moment.

This summary is too short for a thorough description of DDs. Instead, we briefly list the properties that makes them attractive for our purposes. A DD is a lossless compression of a list of numbers (that is, the diagram represents the same list but uses only a modest amount of memory. This is analogous to saying that a Boolean formula is a compressed representation of its truth table). Importantly – and this distinguishes DDs from other compression methods – one can perform operations on the data without decompressing the DD first. For example, suppose that we are considering a certain quantum state $|\varphi\rangle$, which may represent the intermediate state during a quantum computation, and suppose that we know which quantum logic gate this algorithm will next apply. We wish to obtain a description of the state $|\psi\rangle$ after the quantum logic gate has been applied. This is relatively straightforward to do if we operate on the state vector; unfortunately, this always takes $\mathcal{O}(2^n)$ time, which prohibits us from applying the state vector method for problems of interesting size (because then n is too large). Instead, given a DD representing the current quantum state $|\varphi\rangle$, there is an algorithm which constructs a new DD representing the state $|\psi\rangle$ of the system after the quantum logic gate has been applied. In many cases, this process is efficient, because we do not need to decompress the DD to an (exponentially long) state vector in order to obtain a description of the state after the gate is applied. Designing such algorithms and the DDs on which they operate is the primary technical objective of this thesis.

The method described above allows us to analyze any quantum circuit in principle;

in practice, however, the DD may become too big to fit in computer memory, or operations on the DD take too much time; when that happens, our analysis has failed. The size of the DD and time taken by operations depends on how complex the quantum state is and on the type of DD used. Again, understanding and addressing these bottlenecks by analyzing existing and designing new DDs and algorithms to operate on them, is the primary technical goal of this thesis, and is arguably the primary goal of research on decision diagrams more broadly.

Indeed, existing decision diagrams could not adequately analyze many quantum algorithms; not even, for example, so-called stabilizer circuits, even though stabilizer circuits can in fact be efficiently simulated and analyzed using other methods. Therefore, in this thesis, we present a novel type of decision diagram, which we call the Local Invertible Map Decision Diagram (LIMDD) which can do everything that previous decision diagrams could do, plus stabilizer circuits, and, as we shall see, much more. By analyzing our design on paper and implementing it in software, we are able to show that our new DD outperforms existing DDs both in theory and in practice, in Chapter 3 and Chapter 4.

There are many different architectures for decision diagrams, both because new architectures improve upon previous ones and because different designs cater to different use cases. The strengths and weaknesses of these DDs are typically assessed using three criteria: *succinctness* (how large is the decision diagram?), *tractability* (which operations on a DD run in polynomial-time?) and *rapidity* (is one DD faster than another?). However, we found that (1) existing methods for quantum simulation had not previously been mapped out along these criteria; and (2) there was no good method to tell which of two data stuctures is more rapid. Therefore, (1) we provide such a map and (2) we introduce just such a method, by which one can tell when one data structure is more rapid than another, in Chapter 5. In the process, we discover a surprising connection between two existing methods; namely, QMDDs are a special type of matrix product state. Moreover, matrix product states are strictly more rapid than QMDDs. Previously, these two data structures had not been compared; our work therefore brings together two directions of research.

In Chapter 6, we examine an old decision diagram called the *Disjoint Support Decomposition Decision Diagram* (DSDBDD). This DD is not a quantum tool; it can be used for classical model checking. Although the DSDBDD had a software implementation, little was known analytically about its expressive power. We fill this gap by showing that it is, in fact, exponentially more succinct than several other decision diagrams.

Lastly, in Chapter 7, we use the *Sentential Decision Diagram* for classical model checking. Here we are given a computer program with a finite number of variables and are asked what the reachable states are. The goal is to build an SDD which represents the set of reachable states of the input program. Our principal challenge in this case is to choose the SDD's *variable tree* ("vtree") well, because the vtree determines the size of the SDD. To this end, we have the opportunity to analyze the program before we start computing the reachable states using decision diagrams. This allows us to tabulate, for each variable, with which other variables it comes into "contact;" e.g., in an assignment statement, several variables are used to compute the value of another. We empirically compare several different heuristics for choosing the vtree based on information about the interaction of the program's variables.

Samenvatting

Kwantumcomputers zijn een voorgesteld, nieuw type computer. Het doel is om sommige berekeningen veel sneller uit te voeren dan nu mogelijk is door gebruik te maken van de unieke fenomenen die kwantumdeeltjes laten zien: superpositie en verstrengeling. Als en wanneer grote kwantumcomputers gebouwd worden, beloven ze een aantal problemen veel sneller te kunnen oplossen, met name in de natuurkunde en scheikunde, waarmee ze potentieel bijdragen aan belangrijke toepassingen zoals betere medicijnen, batterijen en zonnepanelen. Zo dragen ze bij aan de transistie naar groene energie en een duurzame economie. Om deze beloftes waar te maken, hebben we gereedschappen nodig die deze computers en de algoritmes die erop draaien kunnen analyseren, simuleren, compileren en verifiëren. Tegelijkertijd blijven we natuurlijk gereedschappen nodig hebben om klassieke (d.w.z. niet-kwantum) software te compileren, analyseren en verifiëren. Het doel van dit proefschrift is om bij te dragen aan gereedschap voor deze twee toepassingen.

Voordat we onze bijdragen beschrijven, beschrijven we eerst welk probleem we precies trachten aan te pakken, en wat haar zo moeilijk maakt.

- Een algoritme voor een kwantumcomputer bestaat uit een aantal qubits en zogehete *quantum logic gates* die op de qubits worden uitgevoerd. Wanneer we een dergelijk algoritme analyseren, willen we vragen beantwoorden zoals, “Wat is de meest waarschijnlijke output?” en “Hoe waarschijnlijk is het dat de computer in een bepaalde toestand terecht komt?”

  De grootste uitdaging ligt erin dat de toestand van een kwantumcomputer wordt beschreven door een exponentieel lange lijst getallen: Een computer met een n aantal qubits vereist een lijst van 2^n complexe getallen. Deze exponentiele groei maakt het onmogelijk om deze lijst op te slaan zonder gebruik te maken van compressie, wanneer n groot genoeg is.

- Wanneer we een klassiek stuk software willen analyseren, willen we weten of het voldoet aan zijn specificatie. We willen bijvoorbeeld verifiëren dat een programma niet dezelfde resource uitgeeft aan twee verschillende threads. Een uitdaging is dat het aantal mogelijke toestanden waarin het programma zich kan bevinden groeit als 2^n – opnieuw exponentieel in het aantal bits.

In beide gevallen zal een stuk gereedschap slagen het systeem te analyseren wanneer het erin slaagt om te gaan met deze exponentieel grote objecten. Omdat de uitdagingen in het kwantum- en het klassieke geval zo op elkaar lijken, mogen we ons in dit proefschrift toeleggen op deze gedeelde uitdaging. Een krachtig stuk gereedschap kan vervolgens waarde opleveren voor beide toepassingen.

Het gereedschap waar wij ons in dit proefschrift in specializeren heet een *decision diagram* (DD, of beslissingsdiagram). Decision diagrams zijn een datastructuur met een rijke theorie en met veelvuldige toepassingen in verschillende delen van de computerwetenschappen, waaronder het analyzeren van klassieke- en kwantumcomputers.

We houden deze samenvatting te bondig voor een uitgebreide uitleg van hoe decision diagrams werken. In plaats daarvan sommen we kort op wat hen zo aantrekkelijk maakt voor onze doeleinden. Om te beginnen zijn DDs een vorm van verliesvrije datacompressie: een DD representeert een lijst getallen maar gebruikt, onder gunstige omstandigheden, veel minder geheugenruimte (dit is analoog aan zeggen dat een Boolese formule een gecomprimeerde vorm van zijn waarheidstabel is). Wat voor ons van belang is, en wat DDs onderscheidt van andere compressiemethoden, is dat we operaties op de gecomprimeerde data kunnen uitvoeren zonder deze eerst te decomprimeren.

Bijvoorbeeld, de volgende situatie komt een aantal keer voor in dit proefschrift. Stel dat we een kwantumalgoritme aan het analyseren zijn en we hebben een beschrijving van de huidige toestand van de qubits; we noemen deze toestand $|\varphi\rangle$. Nu willen we weten in welke toestand de qubits zich zullen bevinden nadat de volgende *quantum logic gate* is toegepast; deze nieuwe toestand noemen we $|\psi\rangle$. Als we een beschrijving van $|\varphi\rangle$ hebben in de vorm van een lijst van 2^n getallen, dan is het relatief eenvoudig om ook zo'n beschrijving voor $|\psi\rangle$ te vinden. Helaas kost dit exponentieel veel tijd; op deze manier zullen we dus geen grote systemen kunnen analyseren. Daarom gebruiken we niet een lijst maar een decision diagram om de toestand te beschrijven. Om nu dezelfde quantum logic gate toe te passen gebruiken we een algoritme dat als input $|\varphi\rangle$ neemt en als output $|\psi\rangle$ geeft, beide in de vorm van een DD. Dit algoritme is vaak efficiënt omdat de DD van nieuwe toestand $|\psi\rangle$ gebouwd kan worden zonder dat de (kleine) DD van de oude toestand $|\varphi\rangle$ gedecomprimeerd hoeft te worden. Het ontwerpen van zulke algoritmes en de DDs waar ze op werken is het doel van dit proefschrift.

De methode hierboven stelt ons in staat om *in principe* elk kwantumcircuit te analyseren; echter *in de praktijk* zal de DD vaak veel te groot worden om in het geheugen van een computer te passen, of de algoritmes nemen teveel tijd in beslag. Wanneer dat gebeurt, is onze analyse mislukt. Hoe groot een DD wordt, en hoe lang de operaties duren, hangt namelijk af van de kwantumtoestand en van het type DD dat gebruikt wordt. Het begrijpen en verhelpen van deze beperkingen door bestaande DDs en hun algoritmes te analyseren en nieuwe te ontwerpen is daarom het primaïre technische doel van dit proefschrift.

Bestaande DDs konden veel verschillende kwantumalgoritmes niet effectief analyseren. Onder andere zogeheten *stabilizer circuits* vielen buiten de boot, ondanks dat andere methoden deze circuits wél aankonden. In dit proefschrift presenteren we daarom een nieuw type DD, die we de *Local Invertible Map Decision Diagram* (LIMDD) noemen, die alles kan dat voorgaande DDs konden, plus stabilizer circuits, en nog meer. Door ons ontwerp op papier te analyseren en vervolgens in software te implementeren, kunnen we laten zien dat ons nieuwe DD bestaande DDs zowel op papier als empirisch overtreft, in Hoofdstuk 3 en Hoofdstuk 4.

Er zijn veel verschillende ontwerpen voor decision diagrams. Dat komt zowel doordat nieuwe architecturen hebben geleerd van voorgaanden, en omdat verschillende ontwerpen verschillende toepassingen bedienen. De plus- en minpunten van deze DDs worden gemeten met drie criteria: (1) *succinctness* (beknoptheid: hoe groot wordt de DD?); (2) *tractability* (welke operaties op de DD zijn efficiënt, d.w.z. kosten slechts polynomiaal veel tijd); en (3) *rapidity* (hoe snel is één DD ten op zichte van een andere?). Wij stuitten echter op de bevindingen dat (1) bestaande methodes voor simulatie van kwantumcircuits nog niet op deze criteria in kaart waren gebracht; en (2) er geen simpele methode bestond om te bepalen welke van twee DDs de meer *rapid* is. In hoofdstuk 5 van dit proefschrift (1) beoordelen we een aantal veelgebruikte datastructuren en (2) geven we een nieuwe methode om eenvoudig de meer *rapid* van twee datastructuren aan te wijzen. Al doende ontdekken we een verrassend verband tussen twee welbekende datastructuren die uit verschillende onderzoeksvelden komen, namelijk dat QMDDs een speciaal type matrix product state (MPS) zijn; een gevolg is dat MPS meer *rapid* zijn dan QMDDs. Voorheen waren deze twee methoden nog niet vergeleken; ons werk verbindt dus twee onderzoeksrichtingen.

In hoofdstuk 6 bekijken we een oud DD genaamd de *Disjoint Support Decomposition Binary Decision Diagram* (DSDBDD). Dit stuk gereedschap komt niet uit de kwantumwereld; het wordt gebruikt voor klassieke model checking. Hoewel de DSDBDD reeds een softwareimplementatie heeft, was nog weinig bekend over hoe krachtig hij was. Wij vullen deze kennis aan door te laten zien dat hij exponentieel meer *succinct* is dan enkele andere DDs.

Ten slotte gebruiken we in hoofdstuk 7 *Sentential Decision Diagrams* (SDD) voor klassieke model checking. In dit probleem beschouwen we een computerprogramma en worden we gevraagd welke toestanden bereikbaar zijn vanuit de begintoestand. Ons doel is om een SDD te bouwen die de verzameling bereikbare toestanden weergeeft. De grootste uitdaging is om de *variabelenboom* (vtree) goed te kiezen, omdat die bepaalt hoe groot de SDD wordt, en dus of deze past in het geheugen van een computer (en dus of de analyse lukt of niet). Hiertoe hebben we de gelegenheid om het inputprogramma te analyseren voordat we beginnen met simuleren. We kunnen dus opsommen welke variabelen interactie met elkaar hebben en zo beslissen welke variabelen dus idealiter dicht bij elkaar in de variabelenboom moeten komen. We presenteren de eerste heuristieken om een variabelenboom te kiezen en vergelijken empirisch hoe effectief deze verschillende heuristieken enkele systemen analyseren. De resultaten wijzen erop dat SDDs compacter maar trager zijn dan BDDs.

Curriculum Vitae

Lieuwe was born on 14th October 1993 in Leiderdorp. At Leiden University, he obtained his BSc. degree in Computer Science in 2014 and his MSc. degree in Computer Science in 2018. In 2019, he received the VERSEN Master Thesis Award for his MSc. thesis, supervised by Andre Deutz, entitled “A Quantum Polynomial Hierarchy and a Short Proof of Vyalyi’s Theorem.” At Leiden University and under the supervision of Holger Hoos, Alfons Laarman, and Tim Coopmans he conducted the research presented in this doctoral thesis. During his PhD period, he continued to follow various courses, such as on abstract algebra, model checking, the theory of algorithms, communication in science, academic writing and scientific conduct. In 2020, he received the SETTA Best Video Presentation Award for his presentation of his publication “Model Checking With Sentential Decision Diagrams,” which became Chapter 7 in this thesis. In 2022 and 2023, he made research visits to the Johannes Kepler Universität in Linz, Austria, to collaborate on research on decision diagrams in the group of Robert Wille.

His research interests lie in theoretical computer science, particularly computational complexity theory and algorithm design.

Currently he works at CGI as a software engineer, exploring ways in which industry can adopt newly emerging opportunities in quantum computing.

In his free time, he enjoys playing the cello and doing gymnastics.

Titles in the IPA Dissertation Series since 2022

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN*. Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways.* Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering.* Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty.* Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them.* Faculty of Mathematics and Natural Sciences, UL. 2025-05