Copyright
by
Meghana Aparna Sistla
2026

The Dissertation Committee for Meghana Aparna Sistla
certifies that this is the approved version of the following dissertation:

Succinct Function Representations for Simulation and Verification

Committee:

Swarat Chaudhuri, Supervisor

Kenneth L. McMillan

Warren A. Hunt, Jr.

Thomas W. Reps

Succinct Function Representations for Simulation and Verification

by
Meghana Aparna Sistla

Dissertation

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin
May 2026

Dedicated to my family, with love & gratitude.

Acknowledgments

They say it takes a village to raise a child, and much the same can be said of completing a PhD. I have been fortunate to have such a village, and none more central to it than my advisor, Swarat Chaudhuri. Over the course of this journey, I have learned enormously from him, and I hope to carry his way of thinking into the next chapter of my life. His ability to zoom out and see the big picture, and his insistence on always asking why a problem is worth solving, taught me that finding the right question is often just as important as finding the answer. Above all, Swarat is a good human being, and I have come to believe that this is the most important quality an advisor can possess. He supported every direction I wanted to pursue, offering the right guidance at the right moment. Through multiple rejections and moments of self-doubt, he never stopped encouraging me to follow my interests and trust in my work. For that, I will always be deeply grateful.

I would like to take this opportunity to thank Tom Reps, someone who has been profoundly influential in my life. I feel incredibly fortunate to have worked with him for eight years, and his decision to take me on as a research intern was, without question, one of the best things that has happened to me. He made me feel at home in the United States and gave me my very first taste of what research could be. I still remember him lending me his spare bike on my very first day in Madison, a gesture that showed me that, beyond being a great researcher, he is also a genuinely kind soul. From Tom, I learned so much: how to conduct rigorous research, how to attend to the finest details and yet communicate them with clarity and simplicity, and how to draw meaningful connections across different research areas. Most importantly, he taught me that patience is not just a virtue — it is a cornerstone of a research career. Having worked alongside him and shared so many conversations over these eight years, I will deeply miss his guidance and his stories — from memories of his graduate school days to the cold winters of Madison.

I would also like to thank my committee members: Ken McMillan, who taught me the fundamentals of verification and model checking and built in me a strong foundation in this area, and Warren Hunt Jr., who graciously agreed to join the committee on short notice. I would also like to thank Richard Lipton for suggesting the application of CFLOBDDs to quantum simulation, and Patrick Emonts for his advice on the proper approach to quantum-circuit simulation with tensor networks. I am grateful to Alfons Laarman for helping us identify certain properties of CFLOBDDs that allowed us to make the presentation more precise and clear, and to all the referees of the papers included in this thesis for their thoughtful suggestions on improving the presentation. I would also like to thank the S.N. Bose Scholarship program for its generous support, which in part made this work possible.

I have also been fortunate to have a wonderful set of mentors along the way. My undergraduate advisor, Krishna Nandivada, always believed in me and encouraged me to pursue a PhD — a nudge that set everything in motion. In the years since, Eddie Aftandilian has been a steady presence, always willing to listen and offer grounded, objective insights whenever I needed them. Alongside my academic journey, my time in industry proved to be equally formative. My internship with the Network Verification team at Google in the summer of 2024 introduced me to two wonderful mentors, Ali Kheradmand and Steffen Smolka, who created an engaging and fun learning environment that made me eager to return for a second internship the following summer. I am also grateful to José Cambronero, who responded to my cold emails and opened the door to working with a remarkable group of people — Satish Chandra, Gogul Balakrishnan, Patrick Rondon, Michele Tufano, and José himself. These internships added a richness to this journey that I will always cherish.

Back on campus, I was equally lucky to be surrounded by a wonderful group of labmates who made the day-to-day of a PhD both bearable and enjoyable. My conversations with Josh Hoffman, Atharva Sehgal, Amitayush Thakur, Chenxi Yang, George Tsoukalas, Dweep Trivedi, Eric Hsiung, Thomas Logan, and Rahul Saha,

spanning day-to-day research to life and everything in between, truly made this experience worthwhile. Josh, Atharva, and I were among the few who continued coming into the lab during Covid, which also marked the beginning of my PhD, an experience that was formative in its own right. Amitayush was instrumental in helping me navigate the space of AI, offering his time and knowledge with great generosity, and for that, I remain sincerely grateful. I am also grateful to my administrator, Megan Booth, and program coordinator, Gabrielle Bouzigard, who were always prompt in addressing my queries and made navigating the administrative side of graduate school remarkably smooth.

This journey has been made easy by a wonderful set of people whom I am fortunate to call friends. Sravan, whom I met for the first time at the airport, boarding a flight together to begin our PhDs here, despite having been at the same undergraduate institution for five years. He has become a close confidant with whom I have shared most of my PhD journey. He is a genuinely good friend who cheered for me, supported and advised me through difficult times, pampered me with food, games, and movies, and helped me sail through this journey with ease. He also made sure his friend circle became mine, so that I never felt alone in a foreign country. Nikitha and Sowjanya, who stood by me through my undergraduate years, have continued to be a source of support well into graduate school. Nikitha and I spent two years of grad school as roommates, cooking together, watching movies and countless shows, and she patiently heard me talk about my work daily — sitting through the same presentation more times than I can count and still managing to give me the most honest and useful feedback. I am truly grateful to have both of them in my life. Rachit and Parikshit, having started grad school before us, went out of their way to make sure we felt comfortable, from offering advice to driving us around, and it is thanks to them that we settled into this new place as quickly as we did. The friendships that followed only added to the joy of this journey. The friendsgiving dinners, hikes, board game nights, and countless conversations with Gopi, Sharath, Aniruddh, Advait, Ananya, Nihal, Ankit, and Pallavi ensured there was always as much fun as there was work. I

am certain I have inadvertently left out many others who have equally enriched this journey, and I am grateful to each of them.

Finally, I would like to thank my family. My parents, Nagesh and Sailaja, and my brother, Pranav, have been my steadfast pillars of support throughout this journey, encouraging me to pursue my PhD and standing by me through every moment of self-doubt. My mother, in particular, listened to my highs and lows every single day and always found a way to give me exactly the right advice. This PhD is as much hers as it is mine. My father, with his calm and grounding presence, helped me look at this journey not just emotionally but also rationally, always lending a patient ear when I needed it most. My brother has simply been my greatest source of happiness throughout my life. My happiness and success owe much to my grandparents, Lakshmi Kanta Rao and Sita Maha Lakshmi, whose prayers, warmth, and joy on every call and every visit have been a quiet but constant source of strength throughout this journey. I would also like to thank my in-laws, Subrahmanyam and Indira, and my sister-in-law Vidushi, who, though they joined this journey in its latter half, have been unwavering in their support and encouragement.

I cannot finish without thanking my husband, Vasistha, who came into my life during the second half of this journey. Although he never quite signed up for the difficulties of a PhD, he embraced every bit of it with love and patience, willingly becoming my punching bag on the hard days, cheering me on in the good ones, and through it all remaining my greatest source of comfort. I am so grateful to have had him by my side.

Abstract

Succinct Function Representations for Simulation and Verification

Meghana Aparna Sistla, PhD
The University of Texas at Austin, 2026

SUPERVISOR: Swarat Chaudhuri

Efficient representation of data encoded as Boolean functions—used to model matrices, relations, etc.—is fundamental to several areas of computer science. Binary Decision Diagrams (BDDs) provide a compact representation of such functions and have become a cornerstone of symbolic methods in verification, program analysis, and, more recently, simulation of quantum circuits on classical hardware. However, many BDD-based applications suffer from a pronounced intermediate-size swell: although the initial and final BDDs may remain compact, intermediate representations generated during a computation can grow substantially larger and, in the worst case, increase exponentially in size. Consequently, this size-explosion phenomenon often limits the applicability of BDD-based techniques to problems involving only a few hundred Boolean variables.

To address this limitation, we introduce hierarchy into decision-diagram representations, enabling greater sharing of substructures than is possible with traditional BDDs. The central theme of this dissertation is that embedding hierarchical structure within decision diagrams allows recurring patterns in Boolean and pseudo-Boolean functions to be captured systematically, resulting in more scalable and efficient representations.

Toward this goal, we propose *Context-Free-Language Ordered Binary Decision Diagrams* (CFLOBDDs), which can represent Boolean functions exponentially more succinctly than BDDs, and double-exponentially more succinctly than decision trees, in the best case. They have the potential to permit applications to (i) execute much faster, and (ii) handle much larger problem instances than has been possible heretofore. We applied CFLOBDDs in quantum-circuit simulation, and found that for several standard problems the improvement in scalability, compared to BDDs, is quite dramatic. With a 15-minute timeout, the number of qubits that CFLOBDDs can handle are 65,536 for GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover's Algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

We investigated to what extent linear structure—as well as hierarchical structure—contributes to the ability of CFLOBDDs to support compact representations of Boolean functions. We found that both linear and hierarchical structures are essential for achieving compact function representations. However, CFLOBDDs enforce a balanced partitioning of variables, which can make the representation overly rigid. To address this limitation, we introduced *Generalized CFLOBDDs* (GCFLOBDDs), which support arbitrary input grammars and thereby offer greater structural flexibility. We compared GCFLOBDDs with CFLOBDDs and BDDs on combinational circuits and synthetic benchmarks. The results show that GCFLOBDDs instantiated with suitable custom grammars can perform better than BDDs on many benchmarks (in terms of representation size), and consistently outperform CFLOBDDs: achieving up to 99.9% reduction in representation size, and execution-time improvements of up to 4.59×.

We further extended CFLOBDDs by introducing edge weights, yielding *Weighted Context-Free-Language Ordered BDDs* (WCFLOBDDs), a weighted decision diagram analogous to Weighted BDDs, but with hierarchical structure. WCFLOBDDs can be exponentially more succinct than WBDDs and CFLOBDDs, in the best case. Compared to CFLOBDDs, WCFLOBDDs are particularly useful when

the output domain contains many distinct values. Applied to quantum-circuit simulation, WCFLOBDDs outperform WBDDs on several benchmarks, handling up to 64× more qubits (and up to 128× more than CFLOBDDs) within a 15-minute timeout. These results suggest that WCFLOBDDs combine the strengths of weighted and hierarchical representations, offering improved scalability for this domain.

Finally, we created Quasimodo, an easily extensible, open-source Python library for *symbolic simulation* of quantum circuits. It allows simulations of quantum circuits, checking properties of the outputs of quantum circuits, and debugging quantum circuits. It also allows the user to choose from among several symbolic data-structures—CFLOBDDs, BDDs, WBDDs, and WCFLOBDDs—and can be easily extended to support other symbolic data-structures.

Table of Contents

List of Tables 16
List of Figures 18
Chapter 1: Introduction 23
1.1 Introducing Hierarchy for Improved Compression 25
1.2 Adding Weights to Representations for Better Compression 29
1.3 QUASIMODO: A Quantum-Simulation Tool 30
1.4 Organization 31
Chapter 2: Background 33
2.1 Binary Decision Diagrams (BDDs) 33
2.1.1 A Family of Examples, Decision Trees, and BDDs 34
2.2 Weighted Binary Decision Diagrams (WBDDs) 39
2.3 Quantum Simulation 42
2.3.1 Advantages of Simulation 44
Chapter 3: CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams 46
3.1 Introduction 46
3.2 CFLOBDDs 48
3.2.1 Matched Paths 50
3.2.2 CFLOBDD Requirements 53
3.2.3 CFLOBDDs Defined, Part I: Basic Structure 54
3.2.4 Encoding H_4 and Other Members of $\mathcal{H}$ with a CFLOBDD . . . 62
3.2.5 Reuse of Groupings and Compression of Boolean Functions . . 64
3.3 Canonicalness 69
3.3.1 CFLOBDDs Defined, Part II: Additional Structural Invariants 69
3.3.2 Canonicity of CFLOBDDs 75
3.4 Pragmatics 76
3.4.1 Hash-Consing of Groupings and CFLOBDDs to Create Unique Representatives 77
3.4.2 Equality Testing for CFLOBDDs and proto-CFLOBDDs 79
3.4.3 Function Caching 80
3.5 A Denotational Semantics 80
3.6 Algorithms on CFLOBDDs 82
3.6.1 Primitive CFLOBDD-Creation Operations 83

3.6.2 Unary Operations on CFLOBDDs 86
3.6.3 Binary Operations on CFLOBDDs 90
3.6.4 Representing Matrices and Vectors using CFLOBDDs 102
3.6.5 Kronecker Product 108
3.6.6 Vector-to-Matrix Conversion 113
3.6.7 Matrix Multiplication 116
3.6.8 Path Counting and Sampling 122
3.7 Relations efficiently represented by CFLOBDDs 129
3.7.1 The Equality Relation EQ_n 131
3.7.2 The Hadamard Relation H_n 134
3.7.3 The Addition Relation ADD_n 136
3.8 Applications to Quantum Algorithms 146
3.8.1 Special Matrices 146
3.8.2 Quantum Algorithms 161
3.9 Evaluation 171
3.9.1 Experimental Setup 171
3.9.2 Benchmarks and Experimental Results 176
Chapter 4: WCFLOBDDs: Weighted Context-Free-Language Ordered Binary Decision Diagrams 191
4.1 Introduction 191
4.2 Weighted CFLOBDDs (WCFLOBDDs) 194
4.2.1 Basic Structure of WCFLOBDDs 194
4.2.2 Canonicity 200
4.2.3 Exponential Succinctness Gaps 206
4.3 Operations on WCFLOBDDs 207
4.4 Evaluation 225
4.4.1 Experiments to Answer RQ1 225
4.4.2 Experiments to Answer RQ2 226
Chapter 5: Symbolic Quantum Simulation with Quasimodo 231
5.1 Introduction 231
5.2 Quasimodo's Programming and Analysis Interface 232
5.2.1 Extending Quasimodo 236
5.3 Symbolic Simulation 237

Chapter 6: Do CFLOBDDs Actually Make Use of Linear Structure? 239
6.1 Introduction 239
6.1.1 Nested Words and Nested-Word Automata 241
6.2 Tree-Automata Inspired Decision Diagrams (TIDDs) 245
6.2.1 Basic Structure 245
6.2.2 Canoncity 251
6.2.3 Relationship of $L^{\downarrow_f}$ to Proto-CFLOBDDs 256
6.3 Operations using TIDDs 257
6.3.1 Constant Functions 259
6.3.2 Projection Functions 261
6.3.3 Unary Operations 261
6.3.4 Binary Operations 262
6.3.5 Matrix Multiplication 266
6.3.6 Sampling 269
6.4 Understanding TIDDs, CFLOBDDs, and BDDs through examples . . 271
6.4.1 Relations Efficiently Represented by TIDDs 271
6.4.2 Examples Showing the Use of Linearity in CFLOBDDs 273
6.4.3 Exponential Separation between CFLOBDDs and TIDDs . . . 277
6.5 Evaluation 285
Chapter 7: GCFLOBDDs: Generalized Context-Free-Language Ordered Binary Decision Diagrams 288
7.1 Introduction 288
7.2 Generalized CFLOBDDs (GCFLOBDDs) 289
7.2.1 Grammar 290
7.2.2 Basic Structure 294
7.2.3 Comparisions with CFLOBDDs 303
7.3 Operations on GCFLOBDDs 304
7.3.1 Constant Functions 304
7.3.2 Projection Function 307
7.3.3 Unary Operations on CFLOBDDs 307
7.3.4 Binary Operations on GCFLOBDDs 309
7.4 Evaluation 311
7.4.1 RQ1: Can GCFLOBDDs perform better than CFLOBDDs in the case of representing combinatorial circuits? 314
7.4.2 Can GCFLOBDDs represent functions significantly better than CFLOBDDs in terms of space and time? 317

Chapter 8: Related Work . . . 327
8.1 Comparison with BDD variants . . . 327
8.2 Relationship to PDSs, NWAs, etc. . . . 331
8.3 Prior Approaches to Quantum Simulation . . . 332
8.4 Compression of Programs and Compression Principles . . . 333
Chapter 9: Conclusion . . . 336
Appendix A: Details of Notation for CFLOBDDs and their Components . . . 344
Appendix B: Lexicographic-Order Proposition of CFLOBDDs . . . 346
Appendix C: Proof of the Canonicalness of CFLOBDDs . . . 351
Appendix D: Pair Product Canonicity Proof for CFLOBDDs . . . 365
Appendix E: Additional Operations on CFLOBDDs . . . 368
E.1 Ternary Operations on CFLOBDDs . . . 368
E.2 Restrict . . . 370
E.3 Existential Quantification . . . 370
Appendix F: Boolean Operations via ITE . . . 376
Appendix G: Kronecker Product with CFLOBDDs . . . 377
Appendix H: Efficient Construction of $Column1Matrix_n$ with CFLOBDDs . . . 380
Appendix I: Algorithm for Constructing the CNOT Matrix using CFLOBDDs 384
Appendix J: Construction of Additional Quantum Gates . . . 391
J.1 Controlled-Phase Gate . . . 391
J.2 Swap Gate . . . 393
Appendix K: Time Complexity of Reduce . . . 401
Appendix L: Constructing a Canonical WCFLOBDD from a Decision Tree . . . 412
Appendix M: Algorithms for WCFLOBDDs . . . 417
M.1 Pointwise Multiplication . . . 417
M.2 Pointwise Addition . . . 417
M.3 Kronecker Product . . . 417
M.4 Matrix Multiplication . . . 417
M.5 Sampling . . . 417
M.5.1 Weight Computation . . . 419
M.5.2 Sampling . . . 425
Bibliography . . . 432

List of Tables

3.1 List of operations on CFLOBDDs; *vars* denotes the number of Boolean variables ($= 2^k$, where k is the number of levels of the CFLOBDD). The size measure $|\cdot|$ counts the number of groupings, vertices, and edges—with no double-counting of shared groupings due to hash-consing. In the column for the time complexities of BDD operations, an occurrence of c refers to a BDD argument of the operation, and $|c|_B$ denotes the size of BDD c (the number of nodes and edges). For quasi-reduced BDDs, the time to construct the analog of NoDistinctionProtoCFLOBDD is $\mathcal{O}(2^k)$. Note that the complexity of MatrixMultiply is in terms of the sizes of matrices represented by c_1 and c_2 and not the sizes of c_1 and c_2. 84

3.2 Performance of CFLOBDDs against BDDs, BDDs with dynamic reordering, and SDDs on the synthetic benchmarks for different numbers of Boolean variables. (For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1GB.) 177

3.3 Performance of CFLOBDDs against BDDs for increasing number of qubits. 180

3.4 Table (cont.) of the performance of CFLOBDDs against BDDs for increasing numbers of qubits. 181

3.5 Performance of CFLOBDDs against Quimb on quantum benchmarks for different numbers of qubits. 187

3.6 Table (cont.) of the performance of CFLOBDDs against Quimb on quantum benchmarks for different numbers of qubits. 188

3.7 Table of the performance of CFLOBDDs against BDDs on different combinatorial circuits. 189

4.1 List of operations on WCFLOBDDs, WBDDs and CFLOBDDs; *vars* denotes the number of Boolean variables ($= 2^k$, where k is the number of levels of the WCFLOBDD). The size measure $|\cdot|$ counts the number of vertices and edges—with no double-counting of vertices and edges in groupings that are shared due to hash-consing. In the column for the time complexities of WBDD operations, an occurrence of c refers to a WBDD argument of the operation, and $|c|_B$ denotes the size of WBDD c (the number of nodes and edges). Note that the complexity of MatrixMultiply is in terms of the sizes of the matrices represented by c_1 and c_2, and not the sizes of the data structures c_1 and c_2, plus the cost of an added "final call to `Reduce`." The latter cost depends on the sizes of the structures that are input to and output from `Reduce`. That is, the complexity of $c' = \texttt{Reduce}(c)$ is $\mathcal{O}(|c'| \times |c|)$. Because it is difficult to combine the complexity of the symbolic matrix-multiplication computation and that of `Reduce`, we broke out the cost of `Reduce` as a separate item. 208

6.1 Performance of CFLOBDDs and TIDDs on quantum benchmarks—*GHZ*, *BV*, *DJ*. . . . 286

7.1 Comparison of GCFLOBDDs with CFLOBDDs. . . . 303
7.2 Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits. Note that the rows where the grammar is labeled "Balanced" show the performance of the GCFLOBDD implementation when instantiated with a grammar that makes them equivalent to CFLOBDDs. . . . 313
7.3 Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits (cont.). . 314
7.4 Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits (cont.). . 315
7.5 Table of the performance of Custom-GCFLOBDDs (the grammar for each benchmark is the best grammar that efficiently represented the circuit in terms of the size among the ones shown in Tabs. 7.2–7.4) against BDDs on different combinatorial circuits. . . . 316
7.6 Performance of GCFLOBDDs with two grammar options – a custom grammar and a balanced grammar – and CFLOBDDs, on synthetic benchmarks with different number of variables. The Balanced grammar option simulates CFLOBDD using GCFLOBDDs. . . . 318
7.7 Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.). . . . 319
7.8 Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.) . . . 324
7.9 Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.) . . . 325
7.10 Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.) . . . 326

List of Figures

1.1 Figure shows BDD and CFLOBDD representations for the function $\lambda x_0, x_1, x_2, x_3.(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$. The colored regions and lines indicate the corresponding regions in BDD and CFLOBDD. 26

2.1 H_2 and H_4, the first two members of the family of Hadamard matrices $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$. 34

2.2 Decision trees and BDDs for H_2 and H_4, with plies in interleaved most-significant-bit order—$\langle x_0, y_0 \rangle$ and $\langle x_0, y_0, x_1, y_1 \rangle$, respectively. The bold paths show the assignments $[x_0 \mapsto F, y_0 \mapsto T]$ (for $H_2[0,1]$) and $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$ (for $H_4[0,3]$), respectively. 36

2.3 WBDDs for the matrices H_2, H_4, and H_8, with x variables for rows and y variables for columns. 40

3.1 (a) CFLOBDD for H_2 using the variable ordering $\langle x_0, y_0 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0,1]$. (b) Guide to the terminology introduced in Defn. 3.1. 49

3.2 (a) Datatypes for Grouping, InternalGrouping, DontCareGrouping, ForkGrouping, and CFLOBDD. (b) The CFLOBDD for H_2 (repeated from Fig. 3.1a). (c) An instance of class `CFLOBDD` that represents H_2. 57

3.3 CFLOBDD for the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. (For clarity, some of the level-0 groupings have been duplicated.) . . 61

3.4 Construction of successively larger members of $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$. At level-$(i{+}1)$, each matched path makes two sequential invocations of the level-i grouping (for H_{2^i}), thereby creating $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$. . . 63

3.5 The family of no-distinction proto-CFLOBDDs. 65

3.6 To the left of each arrow, a mock-proto-CFLOBDD that violates the indicated structural invariant; to the right, a corrected proto-CFLOBDD. Invariant violations and their rectifications are shown in red. 73

3.7 CFLOBDD representation of the function $\lambda w, x, y, z.(w \wedge x) \vee (y \wedge z)$, with variable ordering $\langle w, x, y, z \rangle$ 83

3.8 (a) CFLOBDD for $\lambda x_0 x_1.x_0$; (b) CFLOBDD for $\lambda x_0 x_1.x_1$; (c) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1}.x_i$, when $0 \leq i < 2^{k-1}$; (d) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1}.x_i$, when $2^{k-1} \leq i < 2^k$. 88

3.9 Illustrating how $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$ results in $\lambda x_0, x_1.T$. 91

3.10 Why a $\sqrt{P} \times \sqrt{P}$-decomposition is the natural problem decomposition for divide-and-conquer algorithms on structures represented as CFLOBDDs. 105

3.11 (a) and (b) Level-k CFLOBDDs for arrays W and V, respectively; (c) level-$k+1$ CFLOBDD for $W \otimes V$. 109
3.12 131
3.13 Decision diagrams for the $carry\text{-}in = 0$ and $carry\text{-}in = 1$ cases of ADD_n 137
3.14 CFLOBDD representation for the $carry\text{-}in = 0$ case of ADD_n 137
3.15 CFLOBDD representation for the $carry\text{-}in = 1$ case of ADD_n 138
3.16 Recursive CFLOBDD structures for the (a) $carry\text{-}in = 0$ and (b) $carry\text{-}in = 1$ cases of ADD_n. To reduce clutter, a B-connection to a NoDistinctionProtoCFLOBDD is depicted as a straight dashed line to the X exit vertex. 138
3.17 (a) The recursive CFLOBDD structure of the ADD_n relation. Because the 0 and 1 exit vertices of the top-level grouping are associated with the single terminal T (while X is mapped to F), the 0 and 1 exits must be coalesced (indicated in red), which propagates to the internal groupings of the CFLOBDD. (b) The propagation of coalesced exit vertices. 139
3.18 One-hot vector as the yield of a decision tree. The single occurrence of 1 is at the leaf indexed by i—i.e., at the end of the path from the root that follows the bits of i's representation in binary. 150
3.19 The different cases of the CNOT construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`; ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). CNOT takes 3 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, and j for the controlled-bit, where $0 \leq i < j < n$. i' and j' denote bit indices adjusted according to the level l: $i' = i - 2^{l-1}$; $j' = j - 2^{l-1}$. A black square indicates that a particular index is outside the grouping's index range. 153
3.20 The different cases of the CNOT construction, continued. Figures (b) and (c) show the two base cases at level 1, for $CNOT(n, 1, \blacksquare)$ and $CNOT(n, \blacksquare, 1)$, respectively. 154
3.21 CFLOBDD representation of $CNOT_n = CNOT_2^{\otimes n}$ with variable ordering $\langle x_1, y_1, x_2, y_2, \cdots, x_n, y_n \rangle$. (a) Base case (level 2): $CNOT_2$ for 2 bits, with the first bit as the control-bit and second bit as the controlled-bit. (b) Grouping structure used for levels greater than 2. 159
3.22 CFLOBDD performance with a timeout of ninety minutes. Note that in (c) the number of Boolean variables is on a $\log\log$ scale. 178
3.23 CFLOBDD execution time (in seconds) vs. number of qubits (on a log scale) for three of the benchmarks. 183
3.24 Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.) 184
3.25 Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.) 185

4.1 (a), (b), and (c) show WCFLOBDDs for the first three matrices in $\mathcal{H}$—H_2, H_4, and H_8—with the variable ordering $\langle x_0, y_0, x_1, y_1, \ldots \rangle$ ($\vec{x}$: row; $\vec{y}$: column). (d) shows the general structure of a WCFLOBDD that represents H_{2^i}. (e) illustrates the constituents of the WCFLOBDD for H_2. 193
4.2 Object diagram of the WCFLOBDD for matrix H_2 (Fig. 4.1(a)). . . 197
4.3 Representations of the vector $V = [1,1,1,1,2,2,2,4,2,2,2,2,4,4,4,8]^T$ with variables $\langle x_0, x_1, x_2, x_3 \rangle$ representing an index into V. 205
4.4 Two representations of POP_4 using (a) WBDDs and (b) WCFLOBDDs. 207
4.5 `ConstantOneProtoWCFLOBDD` and `ConstantZeroProtoWCFLOBDD` for levels 1 and $k > 1$. 210
4.6 Illustration of `PairProduct` of the WCFLOBDDs for matrices H_2 and I_2. 213
4.7 Illustration of how bilinear polynomials over exit vertices of lower-level groupings arise in matrix multiplication. (a) Left argument of $g \times g'$; (b) right argument of $g \times g'$; (c) the level-1 structure that is constructed in the level-1 subproblem g.AConnection $\times$ g'.AConnection. 220
4.8 Execution times and sizes of WCFLOBDDs and WBDDs on synthetic benchmarks. (For B3, the WBDD size is 1 because $H \times H = I$, which is handled as a special case in MQTDD.) 228
4.9 Execution times and sizes for quantum-circuit benchmarks for WCFLOBDDs, WBDDs, and CFLOBDDs. 229
4.10 Execution times and sizes for quantum-circuit benchmarks for WCFLOBDDs, WBDDs, and CFLOBDDs. (cont.) 230
4.11 Circuit widths of the quantum-circuit benchmarks. 230

5.1 An example of a Quasimodo program that performs a quantum-circuit computation in which the final quantum state is a GHZ state with 4,096 qubits. The program verifies that a measurement of the final quantum state has a 50% chance of returning the all-ones basis-state. 233

6.1 (a) Encoding of a grouping's A-connection and B-connections as call transitions, and its return edges as return transitions of an NWA. The grouping is the one used to encode the family of Hadamard matrices $\mathcal{H}$. (b) and (c) Encoding of the two kinds of level-0 groupings as internal transitions of an NWA. 240
6.2 (a) The CFLOBDD for the Hadamard matrix H_4, with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$ (repeated from Fig. 3.4a). The matched path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0,3]$ (with value 1), is shown in bold. (b) The nested word for the path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$. 243
6.3 Tree representation of the variables of a function 245

6.4 The set of trees $\mathcal{T}_{\mathcal{H}}$ representing the assignments of H_2 with two variables x_0, y_0 as the leaves of the tree. 248
6.5 The diagrams show the runs of tree automata (TIDDs) M_{H_2} and M_{H_4} on two trees corresponding to two assignments for the functions (matrices) H_2 and H_4. 248
6.6 Object-oriented representation of TIDD for H_2. 258
6.7 Diagrams showing a proto-CFLOBDD with an AConnection (Fig. 6.7a) and a BConnection (Fig. 6.7b) to proto-CFLOBDDs that partition the space of strings differently; edges not necessary for the discussion are omitted to remove clutter. 274
6.8 The diagrams show proto-CFLOBDDs for $F(2, .)$, $F(4, .)$. `ND` represents `NoDistinctionNode`. 280
6.9 Diagrams show proto-CFLOBDDs for $F(k, s)$ when $s < 2^{k-1}$ and $s \geq 2^{k-1}$. `ND` represents `NoDistinctionNode`. 281
6.10 Diagrams showing proto-CFLOBDDs of h_n at level-$\log(n) + 1$ (Fig. 6.10a) and level-$l + 1$, $l > \log(n) + 1$(Fig. 6.10b). 283

7.1 DAG representation of grammars in Exs. 7.1, 7.2, 7.3, and 7.4. 293
7.2 GCFLOBDD representations for two functions with different grammars. 295
7.3 Object diagram of GCFLOBDD for the function in Fig. 7.2a. 299
7.4 The diagram shows a `NoDistinctionProtoGCFLOBDD` for the grammar shown in Ex. 7.3. The productions associated with every grouping are also highlighted. 305
7.5 The diagram illustrates the actions taken by `ProjectionProtoGCFLOBDD`, where $i > \sum_{p=1}^{p'-1} \mathcal{V}(A_{j_p})$ for $S_k \rightarrow A_{j_1} ... A_{j_p}$. The first $p' - 1$ layers "call" `NoDistinctionProtoGCFLOBDD`; the p'^{th} layer "calls" `ProjectionProtoGCFLOBDD` (highlighted in blue); and the next $l - p'$ layers "call" `NoDistinctionProtoGCFLOBDD`. . . . 310
7.6 Illustration of how PairProduct and Reduce operate on two BDDs that are a part of larger GCFLOBDDs. 312

B.1 348

C.1 Representations of the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. 355
C.2 (a) Decision tree for $\lambda x_0 x_1.x_0$; (b) fully expanded form of the CFLOBDD; (c) CFLOBDD. 358
C.3 The *Fold* trace generated by the application of Construction 1 to the decision tree shown in Fig. C.2a to create the CFLOBDD shown in Fig. C.2c. 358

C.4 The *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Fig. C.2c to create the decision tree shown in Fig. C.2a. 359

E.1 372

H.1 (a) Base case; (b) the recursive structure for general n. 380

J.1 The different cases of the *CP* construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`, and ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). *CP* takes 4 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, where $0 \leq i < j < n$; and θ (which is only used at top level in the CFLOBDD's value tuple). i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range. Figure (h) shows the base case at level 1; the same proto-CFLOBDD is used for both $CP(n, 1, \blacksquare)$ and $CP(n, \blacksquare, 1)$. Continued in Fig. J.2 392

J.2 The different cases of the *CP* construction. Continued from Fig. J.1. 393

J.3 The different cases of the *SWAP* matrix construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`, and ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). *SWAP* takes 4 arguments: n for the number of bits (number of variables will be $2n$) in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, and $State \in \{0, 1, 2, 3, 4\}$ to indicate the current mode of the construction. i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range. 396

J.4 The different cases of the *SWAP* matrix construction, continued. . . . 397

J.5 The different cases of the *SWAP* matrix construction, continued. . . . 398

J.6 Base cases for the construction of the *SWAP* matrix. There are three base cases at level 2 with 2 bits (i.e., 4 Boolean variables) and five base cases at level 1 with 1 bit (i.e., 2 Boolean variables). 399

J.7 Base cases for the construction of the *SWAP* matrix, continued. . . 400

K.1 $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$. The colors of the edges to proto-CFLOBDDs in C' correspond to the edges to the originating proto-CFLOBDDs in C. 402

Chapter 1: Introduction

Many areas of computer science—such as hardware and software verification, logic synthesis, and equivalence checking of combinatorial circuits—require a space-efficient representation of data, as well as space- and time-efficient operations on data stored in such a representation. Many of the tasks in the aforementioned areas involve operations on either (i) Boolean functions, or (ii) non-Boolean-valued functions over Boolean arguments. In some cases, a level of encoding is involved: the data of interest could be decision trees, graphs, relations, matrices, circuits, signals, etc., which are encoded as functions of type (i) or (ii). Boolean functions – (i) and (ii) – can naively be represented using truth tables or decision trees. However, as the size of the function increases – i.e., as the number of variables of the function increases, the sizes of the truth tables and decision trees increase exponentially.

Binary Decision Diagrams (BDDs) Bryant (1986) are one data structure that is widely used for such purposes. A Boolean function in $B_n = \{0,1\}^n \rightarrow \{0,1\}$ is represented in a compressed form as a BDD. BDDs represent Boolean functions using directed acyclic graphs with the nodes of the graph representing the variables of the function and the leaves of the graph representing the yield of the Boolean function. Ordered-BDDs (OBDDs) follow a systematic decomposition of the Boolean function by applying Shannon decomposition under a fixed variable ordering. Reduced-OBDDs (ROBDDs) are BDDs in which the same variable ordering is imposed on the Boolean variables ("Ordered"), and so-called *don't-care* nodes are removed ("Reduced"). All manipulations of these Boolean functions are carried out using algorithms that operate on ROBDDs. For representing pseudo-Boolean functions—functions of type $\{0,1\}^n \rightarrow D$, where D is some suitable space of values—one can use ROBDDs with non-binary-valued terminals, which are called Multi-Terminal BDDs (MTBDDs) Clarke et al. (1993, 1995) or Algebraic Decision Diagrams (ADDs) Bahar et al. (1997). We will refer to ROBDDs/MTBDDs/ADDs generically as BDDs from hereon.

Because of their ability to represent and manipulate Boolean functions efficiently, BDDs are widely used in hardware (such as VLSI design Meinel and Theobald (1998)) and software verification, program analysis, model checking, logic synthesis, and symbolic reasoning. Their canonical nature under a fixed variable ordering enables efficient equivalence checking and has made them a core data structure in many formal-methods tools.

More recently, BDDs and their variants like Network Decision Diagrams (NDDs) have been successfully applied to network verification Yang and Lam (2015); Li et al. (2025) to symbolically capture network structure and packet-processing behavior. Furthermore, weighted extensions of BDDs, such as Weighted Binary Decision Diagrams (WBDDs) Bhuvaneswari et al. (2009), have enabled symbolic quantum simulation by compactly representing quantum states and operators, highlighting the adaptability of decision-diagram-based frameworks across domains.

BDDs can represent Boolean functions exponentially more succinctly than decision trees by sharing common substructures – primarily, sub-DAGs, that correspond to identical sub-functions. Despite this advantage, many BDD-based applications exhibit a pronounced intermediate-size swell: although the initial and final BDDs may remain compact, intermediate representations generated during computation can grow much larger, and—in the worst case—increase exponentially in size. Such blow-ups can significantly increase runtime or exhaust available memory, thereby rendering the computation impractical. Consequently, this size-explosion phenomenon often limits the applicability of BDD-based techniques to problems involving only a few hundred Boolean variables.

In this thesis, we extend the idea of reusing common substructures by introducing hierarchy, enabling more effective sharing of substructures that correspond to identical sub-functions. The theme of the work described in the thesis can be summarized as follows:

By embedding hierarchical structure into decision-diagram representations, one can systematically capture recurring patterns in Boolean and pseudo-Boolean functions, leading to more scalable and efficient representations.

We find that such hierarchical structure is particularly advantageous when representing (families of) functions for which the number of variables grows exponentially (e.g., $F_1 : \{0,1\}^2 \to D$, $F_2 : \{0,1\}^4 \to D$, ..., $F_n : \{0,1\}^{2^n} \to D$). This approach has shown promising results in the domain of quantum simulation. (See §3.9, §4.4, §6.5, and §7.4.)

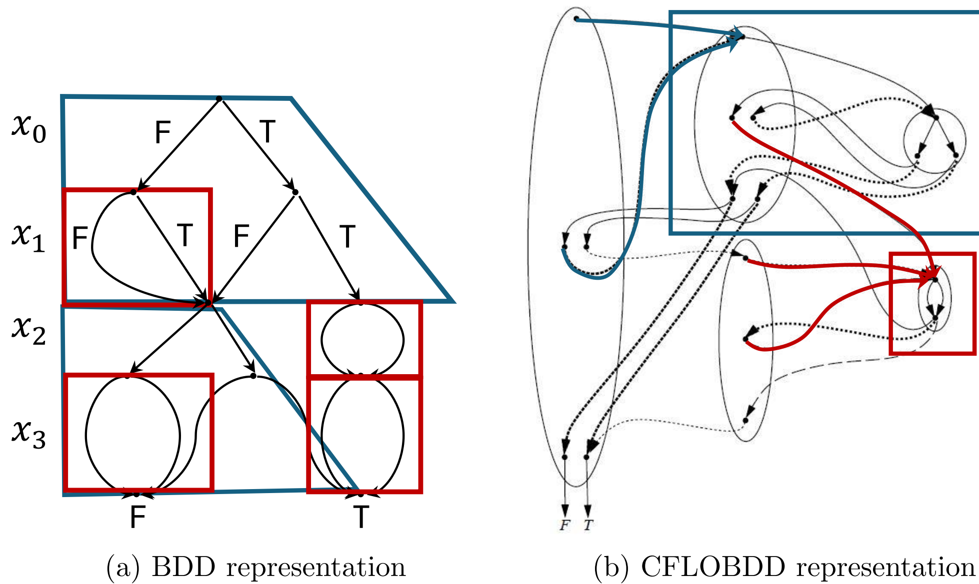
1.1 Introducing Hierarchy for Improved Compression

In this thesis, we introduce *Context-Free-Language Ordered Binary Decision Diagrams* (CFLOBDDs), which can represent Boolean functions exponentially more succinctly than BDDs and double-exponentially more succinctly than decision trees. Whereas a BDD is a bounded-size, branching, but non-looping program, a CFLOBDD can be considered to be a bounded-size, branching, but non-looping program in which a certain form of *procedure call* is permitted.

CFLOBDDs introduce a form of hierarchy absent in BDDs, enabling the capture and reuse of common substructures and thereby yielding what is often a more efficient representation of a Boolean function.[1] More concretely, while BDDs achieve compression by sharing identical sub-DAGs, CFLOBDDs obtain additional compression by sharing intermediate portions of a DAG–effectively reusing "middle-of-a-DAG" structures. Fig. 1.1 shows the representations of BDD (Fig. 1.1a) [2] and CFLOBDD (Fig. 1.1b) for the function $\lambda x_0, x_1, x_2, x_3.(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$ in

[1]Recently, Zhi and Reps (2025) showed that the size of a CFLOBDD for any function h cannot be far worse than the size of a BDD for h—i.e., the bound that relates their sizes is polynomial: if BDD B for function h is of size $|B|$ and uses variable ordering *Ord*, then the size of the CFLOBDD C for h that also uses *Ord* is bounded by $\mathcal{O}(|B|^3)$. They also showed that the bound is tight: there is a family of functions for which $|C|$ grows as $\Omega(|B|^3)$.

[2]Note that the BDD representation here is technically a Quasi-BDD, i.e., the don't care nodes are not reduced. However, our claims in the thesis hold for ROBDDs.



(a) BDD representation (b) CFLOBDD representation

Figure 1.1: Figure shows BDD and CFLOBDD representations for the function $\lambda x_0, x_1, x_2, x_3.(x_0 \wedge x_1) \vee (x_2 \wedge x_3)$. The colored regions and lines indicate the corresponding regions in BDD and CFLOBDD.

four variables. The "middle-of-a-DAG" compression in a CFLOBDD is shown by the blue highlighted region in Fig. 1.1b that corresponds to the blue highlighted regions in the BDD representation where the substructure is repeated twice in a BDD. The introduction of hierarchy and the use of *procedure-calls*, as indicated by the blue edges in Fig. 1.1b, helps reuse the middle of the DAG substructure in CFLOBDDs. The same applies to the red-highlighted substructure as well.

Although not every Boolean function admits such a highly compressed representation, for those that do, CFLOBDDs can be significantly more succinct than BDDs. Consequently, CFLOBDDs have the potential to (i) substantially reduce execution time, and (ii) scale to much larger problem instances than has been possible heretofore. Because CFLOBDDs can potentially represent functions more succinctly than BDDs, they are also less likely to suffer from the intermediate-size swell observed in BDDs (§3.9.2.2 shows this difference in some of our experiments).

The introduction of hierarchy—conceptually akin to a procedure-call

mechanism—enables CFLOBDDs to represent, in the best case, Boolean functions with 2^{2^n} paths using only $\mathcal{O}(n)$ space. This hierarchy is realized through *levels*: a level-n CFLOBDD represents a Boolean function over 2^n variables. At each level, the CFLOBDD structure is decomposed into two components that encode sub-functions over the first and second halves of the variable set. This recursive partitioning facilitates the identification and reuse of common substructures across both halves, thereby enabling further compression. Fig. 1.1b depicts a level-2 CFLOBDD encoding a function over four variables. The smallest ovals represent level-0 structures, the two ovals at the next level correspond to level-1 structures, and the largest enclosing oval represents the level-2 structure.

In Chapter 3 of this thesis, we present theoretical examples demonstrating that CFLOBDDs can achieve exponential succinctness over BDDs. We also describe the CFLOBDD data structure in detail and develop algorithms for performing fundamental operations on CFLOBDDs.

In recent years, Binary Decision Diagrams (BDDs) have been applied to quantum simulation, where quantum circuits are modeled using quantum states (vectors) and quantum gates (matrices). These vectors and matrices grow exponentially with the number of qubits in the circuit, and can lead to scalability issues. We employ CFLOBDDs, which can be exponentially more succinct than BDDs in the best case, to the problem of quantum simulation. This added succinctness enables a more efficient representation of the large matrices and vectors that arise in quantum simulation. We evaluated CFLOBDDs and BDDs on both synthetic benchmarks and quantum-simulation tasks. Performance was compared in terms of representation size and execution time. For problem instances on which both approaches completed successfully, CFLOBDDs were generally more compact and exhibited lower execution times, with the advantages becoming most pronounced near the upper limits of BDD scalability. Moreover, the scalability improvements offered by CFLOBDDs are substantial, both on synthetic benchmarks and in the domain of quantum simulation. With a 15-minute timeout, the number of qubits that CFLOBDDs can handle are

65,536 for GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover's Algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

As described earlier, CFLOBDDs can be viewed as a class of non-recursive hierarchical finite-state machines (HFSMs) with a highly structured form. This naturally raises the question of whether CFLOBDDs can also be characterized as a form of tree automata. In particular, we investigate whether CFLOBDDs fundamentally rely on a linear structure, or whether they are more akin to a restricted class of tree automata that employ hierarchy solely for representational purposes.

To facilitate this comparison, we introduce a new data structure called *Tree-Automata Inspired Decision Diagrams* (TIDDs). TIDDs possess a hierarchical organization similar to that of CFLOBDDs but lack an explicit linear structure, and thus more closely resemble tree automata. In Chapter 6, we study the relative expressive power of CFLOBDDs and TIDDs. We formally describe the structure of TIDDs and develop algorithms for performing fundamental operations on them.

Although TIDDs exhibit exponential separation over BDDs on some of the same families of functions as CFLOBDDs, we provide intuition and evidence that the linear structure inherent in CFLOBDDs enables CFLOBDDs to provide more efficient representations than TIDDs on some classes of functions. In particular, we present an example of a family of functions that demonstrates an exponential separation between CFLOBDDs and TIDDs when the same variable ordering is used. Furthermore, in the domain of quantum simulation, where CFLOBDDs have demonstrated exceptional performance, we observe that TIDDs perform poorly, primarily due to significant blow-ups in the size of intermediate state vectors and unitary matrices. These results demonstrate that CFLOBDDs critically exploit their linear structure for efficient function representation.

In Chapter 7, we investigate the use of alternative decomposition patterns for the structure of CFLOBDDs. In particular, we introduce *Generalized Context-Free-Ordered Binary Decision Diagrams* (GCFLOBDDs), which extend CFLOBDDs by

removing the restriction to a fixed, balanced decomposition. In CFLOBDDs, the decomposition of variables into two halves and the representation of sub-functions over each half can be viewed as following a balanced grammar of the form $X_i \rightarrow X_{i-1}\, X_{i-1}$. In contrast, GCFLOBDDs are designed to operate with arbitrary non-recursive grammars specified by the user.

Allowing decompositions other than the balanced grammar enables GCFLOBDDs to capture repeated sub-functions or structural patterns in Boolean functions that do not naturally conform to a balanced decomposition. In this thesis, we formally define the class of grammars supported by GCFLOBDDs, describe the design of the GCFLOBDD data structure, and develop algorithms for constructing and manipulating GCFLOBDDs with respect to a given input grammar.

We experimentally compare GCFLOBDDs with CFLOBDDs and BDDs on combinatorial circuits and synthetic benchmark functions. Our results indicate that, in certain cases, GCFLOBDDs instantiated with a suitable custom grammar can represent combinatorial circuits more compactly than BDDs. In other cases, BDDs outperform GCFLOBDDs due to the absence of repeated structural patterns in the underlying functions. Nevertheless, even in such scenarios, GCFLOBDDs often achieve better compression than CFLOBDDs. On synthetic benchmarks, GCFLOBDDs with custom grammars outperform CFLOBDDs in representation size, achieving reductions ranging from 47.51% to 99.9%, along with execution-time improvements ranging from 0.78× to 4.59×.

1.2 Adding Weights to Representations for Better Compression

CFLOBDDs, GCFLOBDDs (and BDDs) are data structures that represent pseudo-Boolean functions, with the leaves of the data structures representing the output values of the function. As a result, to represent a function h whose range has a large number of different values V ($V \subseteq D$), the size of the data structure for h

cannot be smaller than $|V|$. To overcome this issue in the case of BDDs, Weighted BDDs were introduced, which are an extension of BDDs with weights on the edges of the data structures. The value of an assignment of h is then computed by "composing" the weights on the path for the assignment.

In Chapter 4, we introduce *Weighted Context-Free-Language Ordered Binary Decision Diagrams* (WCFLOBDDs), which combine ideas from Weighted BDDs (WBDDs) and CFLOBDDs to represent efficiently a broader class of functions than previously possible. We show that, in the best case, WCFLOBDDs achieve an exponential improvement in succinctness over both WBDDs and CFLOBDDs.

We evaluated WCFLOBDDs, WBDDs, and CFLOBDDs on both synthetic benchmarks and quantum-simulation workloads. Our experimental results indicate that WCFLOBDDs outperform WBDDs and CFLOBDDs on the majority of benchmarks. In the quantum-simulation domain, under a 15-minute timeout, WCFLOBDDs can handle circuits with up to 1,048,576 qubits for GHZ states (a 1× improvement over CFLOBDDs and a 256× improvement over WBDDs); 262,144 qubits for the BV and DJ benchmarks (a 2× improvement over CFLOBDDs and a 64× improvement over WBDDs); and 2,048 qubits for QFT (a 128× improvement over CFLOBDDs and a 2× improvement over WBDDs). These results demonstrate that, at least for certain applications, WCFLOBDDs enable the handling of substantially larger problem instances than was previously feasible.

1.3 Quasimodo: A Quantum-Simulation Tool

Given that a variety of symbolic data structures have been used for quantum simulation, we introduce QUASIMODO—a unified quantum-simulation framework—in Chapter 5. QUASIMODO is designed to support multiple symbolic backends, including BDDs, WBDDs, CFLOBDDs, and WCFLOBDDs, while allowing users to write quantum programs independently of the underlying data structure.

QUASIMODO is implemented as a Python library and can be readily imported

to define and simulate quantum circuits. It supports a wide range of quantum gates and operations, including measurement and queries over the probability of specific outcomes. Users can flexibly select the desired symbolic data structure as the backend, and can also extend the system to incorporate additional data structures with minimal effort.

In addition to simulation, QUASIMODO provides basic correctness-checking capabilities. In particular, users can query the set of *all* outcomes whose probabilities exceed a given threshold and verify that these outcomes satisfy a specified assertion.

1.4 Organization

This thesis is organized as follows:

- Chapter 2 reviews background material on Binary Decision Diagrams (BDDs), Weighted Binary Decision Diagrams (WBDDs), and quantum simulation, which is required for understanding the remainder of the thesis.

- Chapter 3 introduces Context-Free-Language Ordered Binary Decision Diagrams (CFLOBDDs), presents algorithms based on CFLOBDDs, establishes exponential separation results with respect to BDDs, and reports empirical results in the domain of quantum simulation.

- Chapter 4 introduces Weighted CFLOBDDs (WCFLOBDDs) and shows that WCFLOBDDs can be exponentially more succinct than both WBDDs and CFLOBDDs, while scaling to larger numbers of qubits in quantum simulation – achieving the best of both worlds.

- Chapter 5 presents QUASIMODO, a tool for quantum simulation based on the techniques developed in this thesis.

- Chapter 6 investigates how the linear structure in CFLOBDDs helps in creating efficient representations of Boolean functions.

- Chapter 7 introduces a generalized grammar-based decomposition for CFLOBDDs, resulting in Generalized CFLOBDDs (GCFLOBDDs), and demonstrates how this framework exploits repeated structure in Boolean functions to achieve more efficient representations than CFLOBDDs.
- Chapter 8 discusses related work.
- Chapter 9 concludes.

Chapter 2: Background

In this chapter, we review some important topics that are needed as background for the research reported on in the thesis. §2.1 discusses Binary Decision Diagrams (BDDs), §2.2 discusses Weighted BDDs, and §2.3 reviews quantum simulation and quantum algorithms.

2.1 Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs) Bryant (1986) are used to represent Boolean functions efficiently. A Boolean function in $B_n = \{0,1\}^n \rightarrow \{0,1\}$ is represented in a compressed form as an ROBDD (Reduced Ordered BDD) data structure. All manipulations of these Boolean functions are carried out using algorithms that operate on ROBDDs. ROBDDs are BDDs in which the same variable ordering is imposed on the Boolean variables ("Ordered"), and so-called *don't-care* nodes are removed ("Reduced"). ROBDDs with non-binary-valued terminals are called Multi-Terminal BDDs (MTBDDs) Clarke et al. (1993, 1995) or Algebraic Decision Diagrams (ADDs) Bahar et al. (1997). We will refer to ROBDDs/MTBDDs/ADDs generically as BDDs from hereon.

In the programming-languages community, BDDs are widely used for program analysis and have been used in Datalog interpreters.

- The SLAM system (later called Static Driver Verifier) was a Microsoft tool for checking temporal properties of device drivers (e.g., that drivers correctly follow API-usage rules) Ball and Rajamani (2001b). BDDs were used in SLAM to represent the abstract transformers of Boolean programs that were abstractions of a driver's source code. BDDs allowed the SLAM developers to increase the capabilities of the IFDS framework for interprocedural dataflow analysis

$$H_2 = \begin{array}{c} \\ \\ x_0 \end{array} \begin{array}{c} y_0 \\ \begin{array}{cc} 0 & 1 \end{array} \\ \begin{array}{c} 0 \\ 1 \end{array} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{array} \qquad H_4 = H_2 \otimes H_2 = \begin{array}{c} y_0 \\ \begin{array}{cc} 0 & 1 \end{array} \\ x_0 \begin{array}{c} 0 \\ 1 \end{array} \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} \end{array} = \begin{array}{c} y_0 y_1 \\ \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ x_0 x_1 \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \end{array}$$

Figure 2.1: H_2 and H_4, the first two members of the family of Hadamard matrices $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$.

Reps et al. (1995) to handle relations over valuations over a Boolean program's Boolean variables Ball and Rajamani (2001a).

- The Datalog solver `bddbddb`, which uses BDDs as the backing representation of relations, was developed by Whaley and Lam to support a variety of program analyses Whaley and Lam (2004); Whaley et al. (2005).
- Lhoták used BDDs in interprocedural program analyses to represent and manipulate collections of large sets, allowing him to use larger programs than previous studies of the factors that affect analysis precision Lhoták (2006).

2.1.1 A Family of Examples, Decision Trees, and BDDs

The theme of this work is compressibility of Boolean functions, for which we need a family of examples that are indexed by some parameter. This section presents the set of *Hadamard matrices* $\mathcal{H}$. It also reviews Boolean functions, decision trees, and BDDs, and shows how decision trees and BDDs can encode the members of $\mathcal{H}$.

Hadamard Matrices The family of Hadamard matrices, $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$, can be defined recursively: for $i \geq 1$, $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$, with H_2 from Fig. 2.1 as the

base case. where $\otimes$ denotes Kronecker product.[1] Fig. 2.1 shows H_2 and H_4, the first two matrices in $\mathcal{H}$. The *Kronecker product* of two matrices is defined as

$$A \otimes B = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} \otimes B = \begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{bmatrix}$$

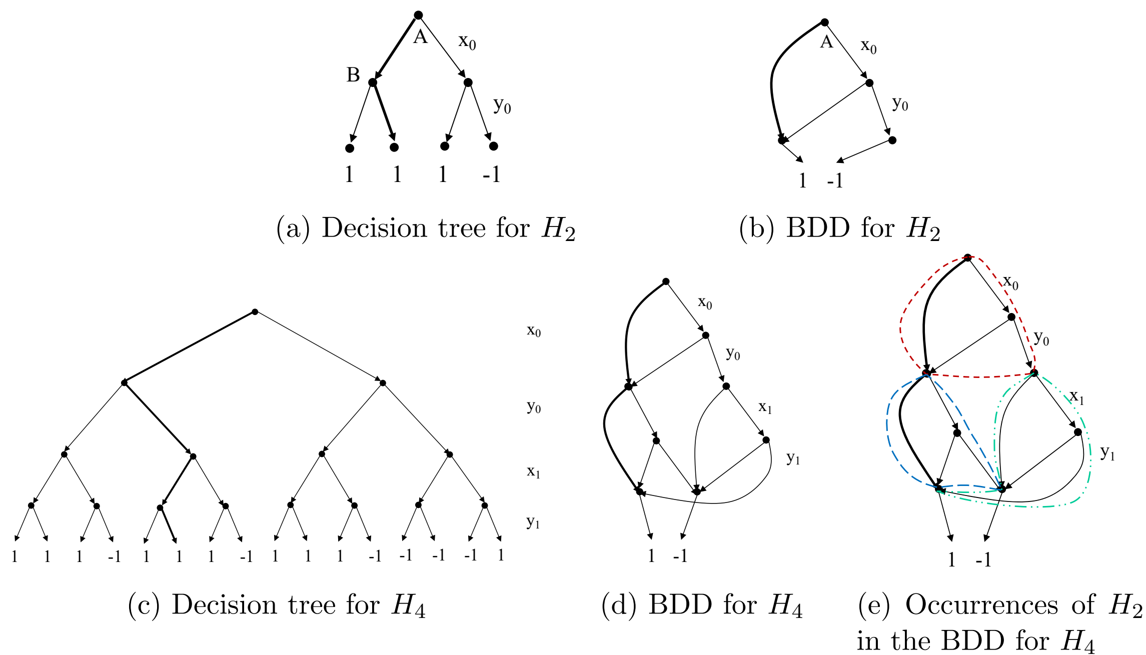
Equivalently, $(A \otimes B)_{ii',jj'} = A_{i,j} \times B_{i',j'}$. If A is $n \times m$ and B is $n' \times m'$, then $A \otimes B$ is $nn' \times mm'$.

For $i \geq 1$, H_{2^i} is a square matrix of size $2^{2^{i-1}} \times 2^{2^{i-1}}$. Thus, the number of rows/columns/entries in $H_{2^{i+1}}$ is the *square* of the number of rows/columns/entries in H_{2^i}. For example, H_4 is 4×4 (16 entries); H_8 is 16×16 (256 entries). An indexing scheme for H_{2^i} can be defined that uses $2^{i-1} + 2^{i-1} = 2 * 2^{i-1} = 2^i$ Boolean variables. As shown in Fig. 2.1, H_2 requires 2 variables—x_0 for the row index and y_0 for the column index—whereas H_4 requires 4 variables—x_0 and x_1 for the row index, and y_0 and y_1 for the column index. In general, H_{2^i} can be treated as a Boolean function of type $\{0,1\}^{2^{i-1}} \times \{0,1\}^{2^{i-1}} \rightarrow \{-1,1\}$. Our convention is that x_0 and y_0 are the most-significant bits of the row and column indexes, respectively; x_1 and y_1 are the next-most-significant bits, respectively, etc.

Boolean Functions A *Boolean function* over n variables is a function in $\{F,T\}^n \rightarrow \{F,T\}$. This work is also concerned with pseudo-Boolean functions: a *pseudo-Boolean* function over n variables and value domain W is a function in $\{F,T\}^n \rightarrow W$. Because there is little chance of confusion, for brevity, we typically refer to such a function as a "Boolean function." We also use 0 and 1 as synonyms for F and T, respectively.

[1] Others use a different indexing scheme: H_2 is the same as with our scheme (as is H_4), but the recursive definition is $H_{2^{i+1}} = H_2 \otimes H_{2^i}$, for $i \geq 1$. Thus, for $i \geq 0$, H_{2^i} is a $2^i \times 2^i$ matrix (and thus has 2^{2i} entries). In contrast, with our indexing scheme, the matrix we call H_{2^i} is a $2^{2^{i-1}} \times 2^{2^{i-1}}$ matrix, for $i \geq 1$ (and thus has 2^{2^i} entries).

Put another way, what we call H_{2^i} would conventionally be known as $H_{2^{2^{i-1}}}$. Not only do we avoid having to write a doubly superscripted subscript, we will see in §3.2.4 that the recursive rule "$H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$" fits particularly well with the internal structure of CFLOBDDs.



(a) Decision tree for H_2 (b) BDD for H_2

(c) Decision tree for H_4 (d) BDD for H_4 (e) Occurrences of H_2 in the BDD for H_4

Figure 2.2: Decision trees and BDDs for H_2 and H_4, with plies in interleaved most-significant-bit order—$\langle x_0, y_0 \rangle$ and $\langle x_0, y_0, x_1, y_1 \rangle$, respectively. The bold paths show the assignments $[x_0 \mapsto F, y_0 \mapsto T]$ (for $H_2[0,1]$) and $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$ (for $H_4[0,3]$), respectively.

Hadamard matrix H_{2^i} can be considered to be a (pseudo-)Boolean function in $\{0,1\}^{2^i} \rightarrow \{-1,1\}$, with some convention about how the 2^i input variables correspond to bits of the row-index and the column-index of the matrix.

Decision Trees A *decision tree* is a tree representation of a Boolean function. For a Boolean function B in $\{F,T\}^n \rightarrow W$, the decision tree T_B for B is a perfect binary tree with n plies and a value from W at each leaf. T_B comes with a specific ordering on the n Boolean inputs of B: each ply of T_B corresponds to some specific Boolean variable v among B's n Boolean input variables. T_B—and hence B—can be evaluated with respect to an input assignment $[v_1 \mapsto b_1, \ldots, v_n \mapsto b_n]$ (where $b_1, \ldots, b_n \in \{F,T\}$) by following a root-to-leaf path in T_B, returning the value that labels the leaf. (Note that v_1 is not necessarily associated with the ply at the root. The order used by T_B is fixed, but can be any of the permutations of the sequence

$\langle v_1, \ldots, v_n \rangle$.)

Figs. 2.2a and 2.2c show two decision trees, with the convention that will be used throughout this work that at each interior node, the left branch is taken when the current Boolean variable in the assignment has the value F (or 0); the right branch is taken for the value T (or 1). Fig. 2.2a shows the decision tree for H_2, which has 2 plies, 3 interior nodes, and 4 leaf nodes, using the variable ordering $\langle x_0, y_0 \rangle$. In Fig. 2.2a, the path highlighted in bold is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$, which corresponds to $H_2[0, 1]$ whose value is 1. Fig. 2.2c shows the decision tree for H_4, which has 4 plies and 15 interior nodes, using the interleaved-variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. The path in bold is for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0, 3]$, whose value is 1.

In Fig. 2.2c, the Kronecker product in the expression $H_4 = H_2 \otimes H_2$ corresponds to *stacking decision trees*. In essence, the $\langle x_0, y_0 \rangle$ plies correspond to the left occurrence of H_2 in "$H_2 \otimes H_2$." At each "leaf" (the four interior nodes after the y_0 ply), there is another copy of H_2 in the $\langle x_1, y_1 \rangle$ plies, with the terminal values labeled with the product of the left H_2's value and the right H_2's value. We can construct a decision tree for each member of $\mathcal{H}$ by repeated stacking, doubling the number of plies each time in accordance with the definition $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$.

Boolean functions and decision trees are related by the following fact:

Observation 2.1. Consider the sets of (i) Boolean functions in $\{0, 1\}^n \to W$, and (ii) n-ply decision trees with leaves labeled by values in W, using a variable ordering that is some fixed permutation of $\langle v_1, \ldots, v_n \rangle$. These sets can be put into one-to-one correspondence.

For each Boolean function $B : \{0, 1\}^n \to W$, create the n-ply decision tree T_B in which the value $B(b_1, \ldots, b_n)$ is placed at the end of the path in T_B for the assignment $[v_1 \mapsto b_1, \ldots, v_n \mapsto b_n]$. Conversely, for each decision tree T_B, let B be the function in $\{0, 1\}^n \to W$ for which $B(b_1, \ldots, b_n)$ equals the value w at the end

of the path in T_B for the assignment $[v_1 \mapsto b_1, \ldots, v_n \mapsto b_n]$. Finally, if two decision trees T_1 and T_2 represent the same Boolean function B, then the sequence of leaves in left-to-right order from each tree are equal, and thus T_1 and T_2 are the same tree (as a mathematical object). Thus, the n-ply decision trees that use a given variable ordering represent the Boolean functions in $\{0,1\}^n \to W$ uniquely.

BDDs A BDD is a compressed representation of a decision tree. Fig. 2.2b shows the BDD for H_2, using the variable ordering $\langle x_0, y_0 \rangle$. Again, left branches are for F (or 0); right branches are for T (or 1). In the H_2 matrix, rows 0 and 1 are different, and hence the BDD node for x_0 is a *fork_node*, which forks to two different substructures. In row 0 of the matrix, columns 0 and 1 are identical, and hence the y_0 ply is skipped in the F branch of x_0, with the F branch of x_0 leading directly to the terminal value 1. Conversely, in row 1 of the matrix, the columns differ, and hence the BDD node for y_0 in the T branch of x_0 is a fork_node. In Fig. 2.2b, the bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0,1]$. (Only the edge for $x_0 \mapsto F$ is highlighted because the ply for y_0 is skipped when $x_0 \mapsto F$.)

Fig. 2.2d shows the BDD for H_4 under the interleaved-variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0,3]$. (The path in the BDD only shows $x_0 \mapsto F, x_1 \mapsto F$ because the plies for y_0 when $x_0 \mapsto F$, and y_1 when $x_0 \mapsto F$ and $x_1 \mapsto F$ are skipped.)

Fig. 2.2e shows that the Kronecker product $H_4 = H_2 \otimes H_2$ corresponds to *stacking BDDs*—in essence, each terminal of the BDD for the left occurrence of H_2 in "$H_2 \otimes H_2$" is replaced by a copy of H_2. The BDD for H_4 contains three occurrences of H_2: one in the $\langle x_0, y_0 \rangle$ plies, and two in the $\langle x_1, y_1 \rangle$ plies. The leftmost $\langle x_1, y_1 \rangle$ occurrence (blue-dashed outline) accounts for the three occurrences of matrix H_2 in the H_4 matrix; the rightmost occurrence (green dashed-double-dotted outline) corresponds to the negated matrix $-H_2$ in the lower-right corner of H_4 (cf. Fig. 2.1). Consequently, one can construct a BDD for each member of $\mathcal{H}$ by repeated stacking,

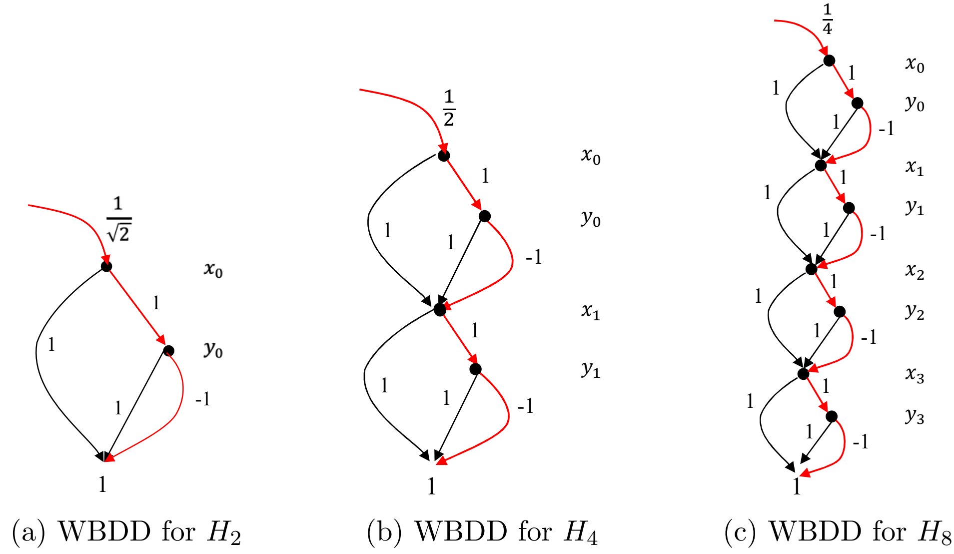
doubling the number of plies each time, per $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$, but only *tripling* the size with each such stacking operation (e.g., $H_8 = H_4 \otimes H_4$ has three copies of H_4, etc.). Consequently, the size of the BDD for H_{2^i} is $O(3^i)$.

Discussion The decision tree for H_{2^i} has height 2^i, 2^{2^i} leaves, and $2^{2^i} - 1$ internal nodes. Thus, the size of the tree is double exponential in i. As observed above, the size of the BDD for H_{2^i} is $O(3^i)$, and hence, compared to decision trees, BDDs achieve *exponential compression* on $\mathcal{H}$.

2.2 Weighted Binary Decision Diagrams (WBDDs)

Multi-terminal BDDs (MTBDDs) Fujita et al. (1997), also known as algebraic decision diagrams (ADDs) Bahar et al. (1997), constitute a natural extension of BDDs for representing functions with non-Boolean ranges. However, they suffer from a fundamental limitation: for a function h whose range contains a large number of distinct values $V \subseteq D$, the size of an MTBDD is necessarily at least $|V|$. Consequently, these representations can become prohibitively large when the output domain is rich. To date, the most effective approach to mitigating this limitation has been the use of *Weighted BDDs* (WBDDs)—BDD-like structures that associate weights with edges—proposed in prior work Niemann et al. (2016); Viamontes et al. (2004).

Consider the family of Hadamard matrices $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$, discussed in §2.1. As i increases, matrices in $\mathcal{H}$ increase in size exponentially. H_{2^i} contains 2^i rows and 2^i columns, and thus 2^{i+1} elements. To index a row (column), $i-1$ row (column) variables are required. Fig. 2.3 shows the WBDD representations for the first three matrices in $\mathcal{H}$: H_2, H_4, and H_8. Every node is associated with a decision variable, and the outgoing edges correspond to the 0 or 1 decision. The x variables represent the row index; the y variables represent the column index. The numbers associated with edges indicate the weight of the edge. The red highlighted paths



(a) WBDD for H_2 (b) WBDD for H_4 (c) WBDD for H_8

Figure 2.3: WBDDs for the matrices H_2, H_4, and H_8, with x variables for rows and y variables for columns.

indicate the all-1 variable-assignment $a = \forall i \in \{1..|\text{vars}|\} : \{x_i \mapsto 1, y_i \mapsto 1\}$, which corresponds to the element in the lower right-hand corner of the matrix. The product of the weights along the path, together with the common weight (at the top of the WBDD) produces the value of the represented function at a. The size of the WBDDs $\{H_{2^i} \mid i \geq 1\}$ increases linearly in the number of variables i.

Semi-Field Weights In WBDDs (and later in WCFLOBDDs–Chapter 4), weights are drawn from a semi-field.

Definition 2.1. Let $\mathcal{D} = \langle A, +, \cdot, \bar{0}, \bar{1} \rangle$ be a set, where $\bar{0}, \bar{1} \in A$, that supports two binary operations: $+$ and $\cdot$. $\mathcal{D}$ is a ***semi-field*** if, for all $a, b, c \in A$, the following properties are satisfied:

Associativity:	$a + (b + c) = (a + b) + c$	Annihilation:	$a \cdot \bar{0} = \bar{0} = \bar{0} \cdot a$
	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	Distributivity:	$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
Commutativity:	$a + b = b + a$		$(b + c) \cdot a = (b \cdot a) + (c \cdot a)$
Identities:	$a + \bar{0} = a = \bar{0} + a$	Mult. Inverse:	$a \neq 0 \Rightarrow \exists a^{-1} \in \mathcal{D} :$
	$a \cdot \bar{1} = a = \bar{1} \cdot a$		$a \cdot a^{-1} = \bar{1} = a^{-1} \cdot a$

The non-$\bar{0}$ elements are a group under multiplication, $\langle A - \{\bar{0}\}, \cdot, \bar{1}\rangle$, whereas $\langle A, +, \bar{0}\rangle$ is only a semi-group. The following properties are consequences of the semi-field axioms:

- The additive identity $\bar{0}$ and multiplicative identity $\bar{1}$ are each uniquely defined.
  - If there is an element $d \in A$ that satisfies $\forall a : a + d = a = d + a$, then $\bar{0} = \bar{0} + d = d$.
  - If there is an element $d \in A$ that satisfies $\forall a : a \cdot d = a = d \cdot a$, then $\bar{1} = \bar{1} \cdot d = d$.

  Consequently, we can speak of $\bar{0}$ as *the* additive identity element and $\bar{1}$ as *the* multiplicative identity element.

- For each element $a \in A$, $a \neq \bar{0}$, the multiplicative inverse is unique.
  - Suppose that b and c are both inverses of a. Then $b = b \cdot \bar{1} = b \cdot (a \cdot c)$ $= (b \cdot a) \cdot c = \bar{1} \cdot c = c$.

- A semi-field has no zero divisors:
  - $a \cdot b = \bar{0} \wedge a \neq \bar{0} \Rightarrow b = \bar{1} \cdot b = (a^{-1} \cdot a) \cdot b = a^{-1} \cdot (a \cdot b) = a^{-1} \cdot \bar{0} = \bar{0}$. Similarly, $b \cdot a = \bar{0} \wedge a \neq \bar{0} \Rightarrow b = b \cdot \bar{1} = b \cdot (a \cdot a^{-1}) = (b \cdot a) \cdot a^{-1} = \bar{0} \cdot a^{-1} = \bar{0}$.

  Equivalently, because $\langle A - \{\bar{0}\}, \cdot, \bar{1}\rangle$ is a group, it is closed under multiplication; thus, for $a, b \in A - \{\bar{0}\}$, $a \cdot b$ can never equal $\bar{0}$.

Because $\mathbb{R}$ and $\mathbb{C}$ are fields, they are also semi-fields. An example of a semi-field that is not a field is the set of invertible $n \times n$ matrices (together with the all-0 matrix of size $n \times n$ as $\bar{0}$), with matrix addition and matrix multiplication.

We need inverse elements with respect to "$\cdot$" to be able to canonicalize WBDDs, as well as weighted decision trees (see Fig. 4.3(b)) and WCFLOBDDs (§4.2.2). Because one multiplies weights as one follows the path for an assignment,

the basic idea is to label left branches (for a Boolean variable bound to 0) with $\bar{1}$, and label right branches (for 1) with some value. Suppose that the branches at a node are originally labeled with a and b, respectively. During canonicalization, the left branch would be labeled with $\bar{1}$, the right branch with $a^{-1} \cdot b$, and the value a would be propagated to every incoming edge. This relabeling maintains the product over all paths—e.g., $\ldots \cdot a \cdot \bar{1} \cdot \ldots$ along the left branch, and $\ldots \cdot a \cdot a^{-1} \cdot b \cdot \ldots$ along the right branch. (There are some special cases for canonicalization when the left edge is originally labeled with $\bar{0}$.) To carry out such weight-propagation steps, "$\cdot$" must be associative, but need not be commutative.

2.3 Quantum Simulation

Quantum algorithms on quantum computers can achieve polynomial to exponential speed-ups over classical algorithms on specific problems. However, to date, there are no practical large-scale implementations of quantum computers. Hence, simulating quantum circuits on classical computers can provide insight on how quantum algorithms perform and scale. [2]

In this context, Binary Decision Diagrams (BDDs) and Weighted Binary Decision Diagrams (WBDDs) have emerged as promising approaches for quantum simulation, where quantum circuits are modeled using quantum states (vectors) and quan-

[2] No knowledge of quantum algorithms is assumed. Everything can be understood in terms of some simple linear algebra. The only subtle concept is that some of the $2^n \times 2^n$ matrices are best thought of in terms of their effect on the *indices* of positions in vectors of size $2^n \times 1$. For instance, the matrix $I \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$ maps the unit vectors $e_{00} = \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $e_{01} = \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $e_{10} = \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, and $e_{11} = \begin{array}{c} 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ to e_{01}, e_{00}, e_{11}, and e_{10}, respectively. In other words, on a unit vector, the effect is to negate the final bit of the index that specifies the position of the 1.

tum gates (matrices). These representations enable symbolic simulation of quantum circuits and have demonstrated strong performance in this domain in recent years.

A single-qubit quantum state can be represented by a pair of complex numbers (i.e., a vector of size 2×1). A quantum state of n qubits can be represented by a complex-valued vector of size $2^n \times 1$; hence, the information capacity increases exponentially with the number of qubits.

A *quantum circuit* takes as input an initial quantum-state vector, and applies a sequence of *quantum gates*, which are each length-preserving transformations, and can be expressed as unitary matrices. Thus, quantum-circuit simulation requires a way to perform linear algebra with vectors of size 2^n and matrices of size $2^n \times 2^n$, where n is the number of qubits involved. Examples of gates that operate on single qubits are $I = \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$, $H = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right]$, and $X = \left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right]$. I leaves a quantum state as is; the Hadamard gate H sends a basis state to a state in "superposition" (i.e., a state that is a non-trivial linear combination of basis states);[3] X complements the indices of a qubit's basis states, and thus flips the positions of the amplitudes, sending $\begin{smallmatrix}0\\1\end{smallmatrix}\left[\begin{smallmatrix}\alpha_0\\ \alpha_1\end{smallmatrix}\right]$ to $\begin{smallmatrix}0\\1\end{smallmatrix}\left[\begin{smallmatrix}\alpha_1\\ \alpha_0\end{smallmatrix}\right]$. Let $M^{\otimes k}$ denote the k-fold Kronecker product of M with itself: $M^{\otimes k} = \overbrace{M \otimes M \otimes \ldots \otimes M}^{k \text{ occurrences of } M}$. The quantum gate that, e.g., applies H to the jth qubit of an n-qubit quantum state is $I^{\otimes(j-1)} \otimes H \otimes I^{\otimes(n-j)}$.

Notation. Generally, when we wish to emphasize the dimensions of objects, a state vector with n qubits (of size $2^n \times 1$) is denoted using a subscript n, and a gate matrix acting on n qubits (of size $2^n \times 2^n$) is denoted using a subscript $2n$. Although the subscripts differ, a vector V_n and matrix M_{2n} have compatible sizes in the matrix-vector product $M_{2n}V_n$.

A completely different subscript convention is used to denote basis vectors: e_s denotes the vector with a one in position s (where s is interpreted in binary) and

[3] A Hadamard gate that operates on a single qubit is the Hadamard matrix H_2 from Fig. 2.1 (§2.1), scaled by $\frac{1}{\sqrt{2}}$ so that it is a unitary matrix.

zeros elsewhere. To be concise, we sometimes use "exponent notation" from formal-language theory to express s. For instance, e_{0^n} and e_{1^n} denote the basis vectors $e_{\underbrace{0\ldots0}_{n\ copies}}$ and $e_{\underbrace{1\ldots1}_{n\ copies}}$, respectively.

Sometimes both conventions come into play. For example, $H_{2n}e_{0^n}$ involves a matrix that acts on n qubits applied to a basis vector with n qubits.

2.3.1 Advantages of Simulation

Simulation of a quantum circuit can have advantages compared to actually running a circuit on a quantum computer:

1. A simulation can deviate from certain requirements of the quantum-computation model and perform the simulation in a way that no quantum device could.
   (a) Some quantum algorithms perform multiple iterations of a particular quantum operator Op (e.g., k iterations, where k is some power of 2). *A simulation can operate on Op itself* (Zulehner and Wille, 2020, Ch. 6), using repeated squaring to create the sequence of derived operators Op^2, Op^4, Op^8, ..., $Op^{2^{\log k}} = Op^k$, which can be accomplished in $\log k$ iterations. The final answer is then obtained using Op^k. A physical quantum computer can only *apply Op sequentially*, and thus must perform k applications of Op.
   (b) The quantum-computation model requires the use of a limited repertoire of operations: every operation is a multiplication by a unitary matrix, and all results (and all intermediate values) must be produced in this way. In contrast, it is acceptable for a simulation to create some intermediate results by alternative pathways.
   (c) In many quantum algorithms, the final state needs to be measured multiple times. When running on a physical quantum computer, part or all of

the quantum state is destroyed after each measurement of the state, and thus the quantum steps must be re-performed before each successive measurement. In contrast, in a quantum simulation the quantum steps need only be performed once, and then *multiple measurements can be made because a (simulated) measurement does not cause any part of the simulated quantum state to be lost*.

Quantum supremacy refers to a computing problem and a problem size beyond which the problem can be solved efficiently on a quantum computer, but not on a classical computer. Quantum simulation is at one of the borders between classical computing and quantum computing: with quantum simulation, a classical computer performs the computation in roughly the same manner as a quantum computer, but can take advantage of shortcuts of the kind listed above. In principle, a more efficient quantum-simulation technique has the potential to change the threshold for quantum supremacy.

2. Current quantum computers are error-prone and can lead to incorrect results, which is not the case with a simulation (modulo bugs in the implementation of the simulation).

3. Quantum simulation has a role in testing quantum computers. In particular, simulation can be used to create test suites for checking the correctness of the output states and measurements obtained from physical hardware.

Chapter 3: CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams

3.1 Introduction

As discussed in §2.1, BDDs are used to represent Boolean functions efficiently. In some applications of BDDs, the initial and final BDD structures are of a reasonable size, but there is an "intermediate swell" during the computation. Such a blow-up can cause operations to take a long time, or cause an application to run out of memory. The size-explosion issue generally limits the use of BDDs to problems involving at most a few hundred Boolean variables.

In this chapter, we introduce a new data structure, called *Context-Free-Language Ordered Binary Decision Diagrams* (CFLOBDDs), which are essentially a plug-compatible replacement for BDDs. CFLOBDDs share many of the good properties of BDDs, but—in the best case—the CFLOBDD for a Boolean function can be *exponentially smaller than any BDD for that function*. Compared with the size of the decision tree for a function, a CFLOBDD—again, in the best case—can give a *double-exponential reduction in size*. Obviously, not every Boolean function has such a highly compressed representation, but for the ones that do, CFLOBDDs offer much better compression than BDDs, and thus have the potential to permit applications to (ii) execute much faster, and (ii) handle much larger problem instances than has been possible heretofore.

Similar to BDDs, CFLOBDDs can represent functions, matrices, graphs, relations, etc. (using either binary-valued or multi-valued terminals, as appropriate), again with the possible advantage of an exponential degree of compression over BDDs. Even for objects that do not fall into the best-case scenario of perfect double-exponential compression, CFLOBDDs may provide better compression than BDDs. Like BDDs, CFLOBDDs are *canonical* (§3.3), and operations are performed on them

directly (§3.6): they are never unfolded to the full decision tree. Moreover, an implementation can ensure that only a single representative is ever constructed for a given function; consequently, the test of whether two CFLOBDDs represent equal functions can be performed merely by comparing the values of two pointers.

CFLOBDDs are based on the following key insight:

> A BDD can be considered to be a special form of bounded-size, branching, but non-looping program. From that viewpoint, a CFLOBDD can be considered to be a bounded-size, branching, but non-looping program in which a certain form of *procedure call* is permitted.

The advantages of this idea are two-fold. First, whereas a BDD of size n can have at most 2^n paths, the "procedure-call" mechanism in CFLOBDDs allows a CFLOBDD of size n to have 2^{2^n} paths (§3.2.5.1). This difference is what lies behind the potential compression advantage of CFLOBDDs. Second, even when best-case compression is not possible, such "procedure calls" allow there to be additional sharing of structure beyond what is possible in BDDs: a BDD can share sub-DAGs, whereas a procedure call in a CFLOBDD shares the "middle of a DAG." (See Figs. 3.1 and 3.4.)

We evaluated CFLOBDDs and BDDs on synthetic benchmarks and for quantum simulation. We compared the performance in terms of size and execution time: on problem sizes for which both approaches ran successfully, CFLOBDDs were generally smaller and had lower execution times, particularly at the upper end of the capabilities of BDDs. Moreover, the improvement that CFLOBDDs bring in scalability is quite dramatic, both for the synthetic benchmarks (§3.9.2.1) and for quantum simulation (§3.9.2.2). We also evaluated CFLOBDDs on hardware, combinatorial circuits §3.9.2.3, and observed that BDDs perform better than CFLOBDDs, showing that when functions do not possess repeated structure across variables, using non-hierarchical structures like BDDs is a better choice.

Organization. §3.2 introduces the basic principles underlying CFLOBDDs. §3.3 introduces some additional structural invariants that allow us to establish that each

Boolean function has a unique, canonical representation as a CFLOBDD. §3.4 discusses how some standard techniques—hash-consing Goto (1974), function-caching (or *memo functions* Michie (1967)), and reference counting—apply to CFLOBDDs. §3.5 gives a denotational definition of the function that a CFLOBDD represents. §3.6 presents algorithms for a variety of CFLOBDD operations. §3.7 demonstrates an exponential gap between CFLOBDDs and BDDs: the CFLOBDD for a function f can be exponentially smaller than any BDD for f. §3.8 turns to the application of CFLOBDDs for quantum simulation, and shows how CFLOBDDs can represent efficiently some special matrices used in quantum algorithms. §3.9 poses two experimental questions and presents the results of experiments on synthetic and quantum-simulation benchmarks. For several problems, the improvement in scalability enabled by CFLOBDDs is quite dramatic. In particular, in the quantum-simulation benchmarks, the number of qubits that could be handled using CFLOBDDs was larger, compared to BDDs, by a factor of 128× for GHZ; 1,024× for BV; 8,192× for DJ; and 128× for Grover's algorithm.

3.2 CFLOBDDs

CFLOBDDs are a binary decision diagram inspired by BDDs, but the two data structures are based on different principles. A BDD is an acyclic finite-state machine (modulo ply-skipping), whereas a CFLOBDD is a particular kind of *single-entry, multi-exit, non-recursive, hierarchical finite-state machine* (HFSM) Alur et al. (2005a). This section describes the basic principles of CFLOBDDs, illustrating them via encodings of H_2 and H_4 with the variable orderings $\langle x_0, y_0 \rangle$ and $\langle x_0, y_0, x_1, y_1 \rangle$, respectively.

Intuition Before discussing the CFLOBDD data structure in detail, we give some intuition about the decomposition principle used in CFLOBDDs.

Consider a function $f : \{0,1\}^n \to [1 \ldots m]$ over variables $x_0, \ldots, x_{n-1}$. In

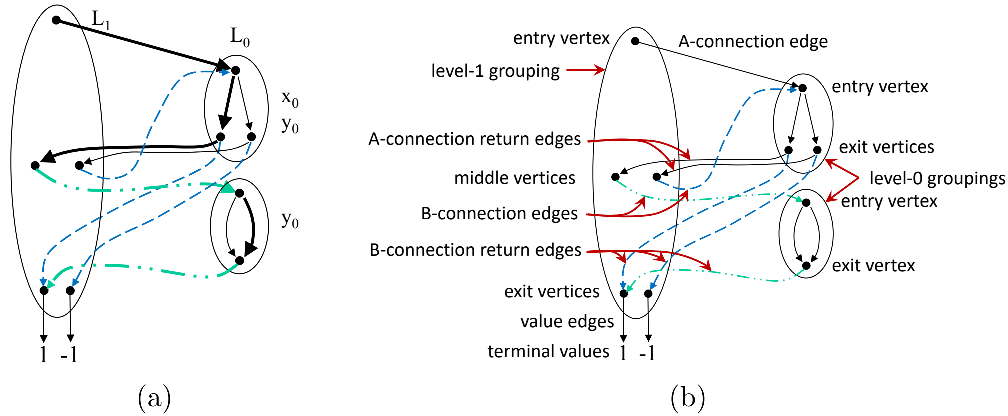


Figure 3.1: (a) CFLOBDD for H_2 using the variable ordering $\langle x_0, y_0 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0, 1]$. (b) Guide to the terminology introduced in Defn. 3.1.

the classical Shannon decomposition of f, one looks at the value of x_0 and then derives two co-factors $g_0 = f|_{x_0=0}$ and $g_1 = f|_{x_0=1}$, both of which are functions over variables $x_1, \ldots, x_{n-1}$. Functions g_0 and g_1 can be combined to yield f by the identity $f = \overline{x_0} \cdot g_0 + x_0 \cdot g_1$ (where $\overline{x_0}$ denotes the complement of x_0, "·" denotes logical-and, and "+" denotes logical-or). (See (Clarke et al., 1995, §4.2) for a precise definition of the generalization of the Shannon decomposition for MTBDDs.) The same decomposition can be carried out recursively on g_0 and g_1, and OBDDs—whether reduced or not—exploit this decomposition by sharing common co-factors that arise in the different plies of the recursive decomposition.

The decomposition used in CFLOBDDs is different. The number of variables n is assumed to be a power of 2, and at each decomposition level the variables are divided into two halves: $x_0, \ldots, x_{n/2-1}$ and $x_{n/2}, \ldots, x_{n-1}$.[1] Let g_0 be the function of the first $n/2$ variables that maps them to $[1 \ldots k]$, where k is the number of equivalence classes

[1]For a Boolean function of m variables that is not a power of 2, one can pad the function with dummy Boolean variables to reach the next higher power of 2. Depending on the function, the user may choose to interleave the dummy variables among the "legitimate" variables or place them all at the end (or some combination of both). By this device, every Boolean function can be represented as a CFLOBDD. (See also the discussion in §3.3.2 of property (2).)

of residual functions one has after the first $n/2$ variables of f are read. (k equals the number of nodes in the corresponding BDD for f at ply $n/2$.) For each $i \in [1 \ldots k]$, g_i is the appropriate function over the remaining $n/2$ variables, which combined with g_0 (based on index i), and an appropriate matching of returned values, yields f.[2] The representation allows sharing across all the functions $g_0, g_1, \ldots, g_k$. Moreover, the divide-the-variables-in-half decomposition is carried out recursively on $g_0, g_1, \ldots, g_k$, with mutual sharing of the decomposed functions that arise at all levels.

Rather than producing a DAG-structured data structure, as one has with BDDs, the divide-the-variables-in-half decomposition leads to a structure that resembles an HFSM (or, alternatively, the interprocedural control-flow graph for a non-recursive, multi-procedure program).

3.2.1 Matched Paths

The CFLOBDD representation of H_2 consists of three *groupings*, shown as three ovals in Fig. 3.1a.[3] Each CFLOBDD grouping is associated with a given *level*. The two small ovals are at level 0 (labeled L_0), and the large oval is at level 1 (labeled L_1). There is an implicit hierarchical structure to the levels, and level-0 groupings are said to be *leaves* of the CFLOBDD. There are only two possible types of level-0 groupings:

- A level-0 grouping like the one at the upper right in Fig. 3.1a is called a *fork grouping*.
- A level-0 grouping like the one at the lower right in Fig. 3.1a is called a *don't-care grouping*.

[2]We are being deliberately vague about how $g_0, g_1, \ldots, g_k$ are combined, because the details are somewhat complicated. See Defns. 3.1 and 3.3 for the precise definition.

[3] Groupings are represented in memory as a kind of node structure, but we will use "nodes" solely for decision trees and BDDs. Groupings are depicted as ovals, and the dots inside will be referred to as "vertices."

The vertex at the top of each grouping is the grouping's *entry vertex*. The entry vertex of a level-0 grouping corresponds to a decision point: left branches are for F (or 0); right branches are for T (or 1). The vertices at the bottom of each grouping are called *exit vertices*; those in the middle of the level-1 grouping are called *middle vertices*.

In matrix H_2, each entry is either 1 or -1. Each assignment over $\langle x_0, y_0 \rangle$ corresponds to a special kind of path in Fig. 3.1a that leads to either 1 or -1. Each such path starts from the entry vertex of the level-1 grouping, making "decisions" for the next variable in sequence each time the entry vertex of a level-0 grouping is encountered.

Fig. 3.1a illustrates the key principle behind CFLOBDDs—namely, the use of a *matching condition on paths*. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$, which corresponds to $H_2[0, 1]$. The path starts at the level-1 grouping's entry vertex and goes to the entry vertex of the level-0 fork grouping via a solid edge (—); takes the left branch of the fork grouping (corresponding to $x_0 \mapsto F$); and leaves the fork grouping via a solid edge (—), reaching the leftmost of the middle vertices of the level-1 grouping. The path then goes to the entry vertex of the level-0 don't-care grouping via a dashed-double-dotted edge (— · · —); takes the right branch of the don't-care grouping (corresponding to $y_0 \mapsto T$); and leaves via a dashed-double-dotted edge (— · · —), reaching the leftmost exit vertex of the level-1 grouping, which is connected to the terminal value 1 (the value of $H_2[0, 1]$). A pair of incoming and outgoing edges of a grouping, such as the pairs of black solid edges and green dashed-double-dotted edges in the bold path in Fig. 3.1a, are said to be *matched*. The bold path itself is called a *matched path*. This example illustrates the following principle:

Matched-Path Principle. *When a path follows an edge that returns to level i from level $i-1$, it must follow an edge that matches the closest preceding edge from level i to level $i-1$.*

Formally, the matched-path principle can be expressed as a condition that—for a path to be *matched*—the word spelled out by the labels on the edges of the

path must be a word in a certain context-free language Yannakakis (1990). (This idea is the origin of "CFL" in "CFLOBDD".) One way to formalize the condition is to label each edge from level i to level $i-1$ with an open-parenthesis symbol of the form "$(_b$", where b is an index that distinguishes the edge from all other edges to any entry vertex of any grouping of the CFLOBDD. (In particular, suppose that there are NumConnections such edges, and that the value of b runs from 1 to NumConnections.) Each return edge that runs from an exit vertex of the level $i-1$ grouping back to level i, and corresponds to the edge labeled "$(_b$", is labeled "$)_b$". Each path in a CFLOBDD then generates a string of parenthesis symbols formed by concatenating, in order, the labels of the edges on the path. (Unlabeled edges in the level-0 groupings are ignored in forming this string.) A path in a CFLOBDD is called a *matched-path* iff the path's word is in the language $L(matched)$ of balanced-parenthesis strings generated by

$$matched \rightarrow \epsilon \mid matched\ matched \mid (_b\ matched\ )_b \quad 1 \leq b \leq \text{NumConnections} \tag{3.1}$$

Only *matched*-paths that start at the entry vertex of the CFLOBDD's highest-level grouping and end at a terminal value are considered in interpreting a CFLOBDD.

In the figures in the paper, we use black solid (—), blue dashed (− − −), red short-dashed (- - - -), purple dashed-dotted (− · · −), and green dashed-double-dotted (− · · −) edges, in the indicated colors, rather than attaching explicit labels to edges. To reduce the number of colors used, we sometimes re-use colors in a given figure; however, it should still be clear which pairs of edges match.

The matched-path principle allows a given grouping to play multiple roles during the evaluation of a Boolean function. In particular, the level-0 groupings are shared, and thus are used to interpret *different variables at different places in a matched path through a CFLOBDD*. For example, the level-0 fork grouping in Fig. 3.1a is used to interpret (i) x_0 (when "called" via the black solid edge), and

(ii) y_0 (when "called" via the blue dashed edge, which happens when $x_0 \mapsto T$).[4] The edge-matching condition is important because the black solid return edges lead to the level-1 grouping's middle vertices, whereas the blue dashed return edges lead to the level-1 grouping's exit vertices.

In Fig. 3.1a, the fork grouping is labeled with x_0 and y_0, and the don't-care grouping with y_0. However, because the level-0 groupings interpret different variables at different places in a matched path, in later diagrams the level-0 groupings are generally not labeled with specific variables. In general, the principle is as follows:

Contextual-Interpretation Principle. *A level-*0 *grouping is not associated with a specific Boolean variable. Instead, the variable that a level-*0 *grouping refers to is determined by context: the nth level-*0 *grouping visited along a matched path is used to interpret the nth Boolean variable.*

The reader might be worried by the fact that Fig. 3.1a contains cycles. That is, if one ignores the ovals in Fig. 3.1a, as well as the distinctions among solid, dashed, and dashed-double-dotted edges, one is left with a cyclic graph: there is a cycle that starts at the rightmost middle vertex of the level-1 grouping, follows the blue dashed edge (−−−) to the entry vertex of the level-0 fork-grouping, takes the right branch, and returns along the black solid edge (—) to the rightmost middle vertex of the level-1 grouping. However, that path is excluded from consideration because it is not a matched path: the solid edge does not match with the preceding dashed edge.

3.2.2 CFLOBDD Requirements

In designing CFLOBDDs, the goal is to meet the following five requirements:

[4] The term "call" is by analogy with how matched paths model the actions of procedure calls in graphs used for interprocedural dataflow analysis Sharir and Pnueli (1981); Reps et al. (1995), interprocedural slicing Horwitz et al. (1990), and model checking hierarchical state machines (Benedikt et al., 2001, §5).

1. *Soundness*: Every level-k CFLOBDD represents a decision tree of height 2^k and size 2^{2^k}
2. *Completeness*: each decision tree of height 2^k and size 2^{2^k} can be encoded as a level-k CFLOBDD
3. *Best-case double-exponential compression*: in the best case, a decision tree of height 2^k and size 2^{2^k} can be encoded as a level-k CFLOBDD of size k
4. *Canonicity*: CFLOBDDs are a canonical representation of Boolean functions
5. *Computational efficiency*: most operations run in time polynomial in the sizes of (i) the input CFLOBDDs, or (ii) the input CFLOBDDs and the output CFLOBDD

These requirements are similar to those for BDDs, but with double-exponential parameters—rather than single-exponential parameters—in Requirements (1)–(3). To satisfy these more stringent requirements, we define a data structure that is quite different from BDDs (see §3.2.3 and §3.3.1).

Requirements (1) and (3) are established in §3.2.3.4 and §3.2.5.2, respectively. Requirements (2) and (4) are established in §3.3 and Appendix §C. Requirement (5) is addressed in §3.4, §3.6, and §K; in particular, Tab. 3.1 at the beginning of §3.6 lists the fourteen main operations on CFLOBDDs and the asymptotic running times of the algorithms that we give for the operations. BDDs enjoy the more desirable property that most operations run in time polynomial in the sizes of the input BDDs, but the same property does not seem possible for CFLOBDDs. §K establishes that the time complexity of a key subroutine used in several of the CFLOBDD operations to maintain canonicity is polynomial in the sizes of the input and output CFLOBDDs.

3.2.3 CFLOBDDs Defined, Part I: Basic Structure

Our formal definition of CFLOBDDs is given in two parts: Defn. 3.1 (below) and Defn. 3.3 (§3.3.1). Defn. 3.1 defines the basic structure of CFLOBDDs, whose various elements are depicted in Fig. 3.1b. Defn. 3.3 imposes some additional structural

invariants to ensure that CFLOBDDs provide a canonical representation of Boolean functions. Much about CFLOBDDs can be understood just from Defn. 3.1, so we postpone introducing the structural invariants until we address canonicity in §3.3. Where necessary, we distinguish between *mock-CFLOBDDs* (Defn. 3.1) and *CFL-OBDDs* (Defn. 3.3), although we typically drop the qualifier "mock-" when there is little danger of confusion. Fig. 3.1b illustrates Defn. 3.1 using the CFLOBDD that represents Hadamard matrix H_2.

Definition 3.1 Mock-CFLOBDD; see Fig. 3.1b**.** A *mock-CFLOBDD* at level k is a hierarchical structure made up of some number of *groupings*, of which there is one grouping at level k, and at least one at each level $0, 1, \ldots, k-1$. The grouping at level k is the *head* of the mock-CFLOBDD.

Each grouping g_i at level $0 \leq i \leq k$ has a unique *entry vertex*, which is disjoint from the set of *exit vertices* of g_i.

If $i = 0$, g_i is either a *fork grouping* or a *don't-care grouping*, as depicted in the upper right and lower right of Fig. 3.1b, respectively. The entry vertex of a level-0 grouping corresponds to a decision point: left branches are for F (or 0); right branches are for T (or 1). A don't-care grouping has a single exit vertex, and the edges for the left and right branches both connect the entry vertex to the exit vertex. A fork grouping has two exit vertices: the entry vertex's left and right branches connect the entry vertex to the first and second exit vertices, respectively.

If $i \geq 1$, g_i has a further disjoint set of *middle vertices*. We assume that both the middle vertices and the exit vertices are associated with some fixed, known total order (i.e., the sets of middle vertices and exit vertices could each be stored in an array). Moreover, g_i has an *A-connection* edge that, from g_i's entry vertex, "calls" a level-i-1 grouping a_{i-1}, along with a set of matching *return edges*; each return edge from a_{i-1} connects one of the exit vertices of a_{i-1} to one of the middle vertices of g_i. In addition, for each middle vertex m_j, g_i has a *B-connection* edge that "calls"

a level-i-1 grouping b_j, along with a set of matching *return edges*; each return edge from b_j connects one of the exit vertices of b_j to one of the exit vertices of g_i.

If $i = k$, g_k has a set of *value edges* that connect each exit vertex of g_k to a *terminal value.*

Fig. 3.1b shows where the concepts from Defn. 3.1 occur in the CFLOBDD that represents Hadamard matrix H_2.

3.2.3.1 An Object-Oriented Pseudo-Code

In later parts of the paper, we state algorithms using an object-oriented pseudo-code. In accordance with the terminology introduced above, the basic classes that are used for representing multi-terminal CFLOBDDs are defined in Fig. 3.2a: `Grouping`, `InternalGrouping`, `DontCareGrouping`, `ForkGrouping`, and `CFLOBDD`. More details about the notation used in our pseudo-code can be found in Appendix §A.

Fig. 3.2c shows how the CFLOBDD from Fig. 3.1a is represented as an instance of class `CFLOBDD`. There are no entry, middle, and exit vertices as such. Instead, a pointer to a `Grouping` object serves as the object's entry vertex. Numbers in the range [1..`numberOfBConnections`] serve as middle vertices, and numbers in the range [1..`numberOfExits`] serve as exit vertices. In the level-1 `InternalGrouping` in Fig. 3.2c, one can see that a `ReturnTuple`—which holds a sequence of return-edge targets—is associated with each outgoing `AConnection` or `BConnection` edge. This organization facilitates implementing the matched-path principle: when a level-l+1 grouping g_1 "calls" level-l grouping g_2, there is an associated `ReturnTuple` rt_1 (stored in g_1); a matched path starting at the entry of g_2 leads to some exit-vertex index i of g_2; and $rt_1[i]$ holds the target in g_1 of the matching return edge.

Similarly, there are no explicit edges in `DontCareGrouping` and `ForkGrouping` objects. Instead, the decision taken at the level-0 grouping's entry vertex selects the appropriate exit-vertex index, which is used to index into a `ReturnTuple` of the "calling" level-1 `InternalGrouping`.

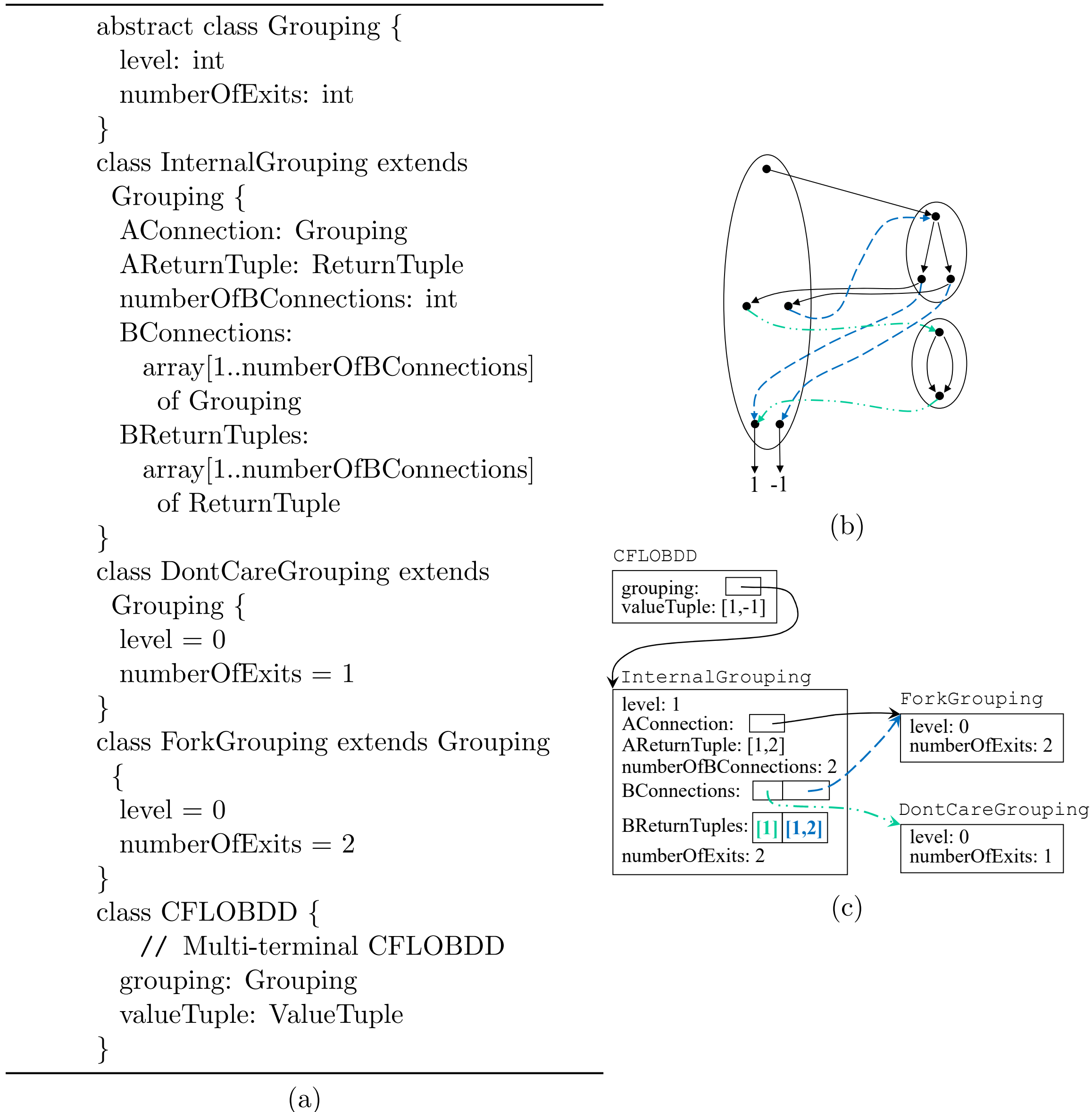


Figure 3.2: (a) Datatypes for Grouping, InternalGrouping, DontCareGrouping, ForkGrouping, and CFLOBDD. (b) The CFLOBDD for H_2 (repeated from Fig. 3.1a). (c) An instance of class `CFLOBDD` that represents H_2.

3.2.3.2 Rationale

Defn. 3.1, Fig. 3.1b, and Fig. 3.2a introduce a substantial amount of new terminology. However, the rationale behind it is really quite simple, and goes back

to the Matched-Path Principle. In particular, each `InternalGrouping` object g at level $i > 0$ represents a family of matched paths. A traversal of a matched path from g's entry vertex to an exit vertex of g uses the fields of g (Fig. 3.2a) in the following order:

$$\begin{array}{lcllll} \textit{matched}(\text{at level } i) & = & \text{AConnection} & \textit{matched}(\text{at level } i\text{-}1) & \text{AReturnTuple}[\cdot] \\ & & \text{BConnection} & \textit{matched}(\text{at level } i\text{-}1) & \text{BReturnTuples}[\cdot] \end{array} \tag{3.2}$$

Note how Eqn. (3.2) mimics the form of the grammar for matched paths from Eqn. (3.1).

3.2.3.3 Inductive Arguments about CFLOBDDs

To be able to make inductive arguments about CFLOBDDs, it is convenient to introduce one additional bit of terminology:

Definition 3.2 Mock-proto-CFLOBDD**.** A *mock-proto-CFLOBDD* at level i is a grouping at level i, together with the lower-level groupings to which it is connected (and the connecting edges). In other words, a mock-proto-CFLOBDD has the following recursive structure:

- a mock-proto-CFLOBDD at level 0 is either a fork grouping or a don't-care grouping
- a mock-proto-CFLOBDD at level i is headed by a grouping at level i whose
  - A-connection edge and associated return edges "call" a level-(i-1) mock-proto-CFLOBDD, and
  - B-connection edges and their associated return edges "call" some number of level-(i-1) mock-proto-CFLOBDDs.

The difference between a proto-CFLOBDD and a CFLOBDD is that the exit vertices of a proto-CFLOBDD have not been associated with specific values. One

cannot argue inductively in terms of CFLOBDDs because its constituents are proto-CFLOBDDs, not full-fledged CFLOBDDs. Thus, to prove that some property holds for a CFLOBDD, there will typically be an inductive argument to establish a property of the proto-CFLOBDD headed by the outermost grouping of the CFLOBDD, with an additional argument about the CFLOBDD's value edges and terminal values.

One example of an inductive argument allows us to establish the number of times $D(i)$ that each matched path in a level-i proto-CFLOBDD reaches a decision vertex—i.e., the entry vertex of a level-0 grouping. In particular, $D(i)$ is described by the following recurrence relation:

$$D(0) = 1 \qquad\qquad D(i) = D(i-1) + D(i-1), \tag{3.3}$$

which has the solution $D(i) = 2^i$.

3.2.3.4 Soundness and an Operational Semantics

Eqn. (3.3) allows us to establish Requirement (1) from §3.2.2. Eqn. (3.3) has the solution $D(i) = 2^i$, so each matched path from the entry vertex of a level-k CFLOBDD passes through the entry vertex of a level-0 grouping exactly 2^k times before reaching a terminal value $v \in V$, for some value domain V. Consequently, each (multi-terminal) CFLOBDD represents a function in $\{T, F\}^{2^k} \rightarrow V$—i.e., the same set of functions that decision trees represent.

We can also use the Contextual-Interpretation Principle to obtain an operational semantics for (mock-)CFLOBDDs, given as Alg. 1. This algorithm is a divide-order-and-conquer algorithm that specifies how to interpret a given CFLOBDD `n` with respect to a given Assignment `a` to the Boolean variables. (We assume that an Assignment is given as an array of Booleans, whose entries—starting at index-position 1—are the values of the successive variables.)

Subroutine `InterpretGrouping` performs a recursive traversal over n, following AConnections, BConnections, and return edges. When a level-0 grouping

Algorithm 1: An operational semantics of CFLOBDDs

```
1  Algorithm InterpretCFLOBDD(n, a)
     Input: CFLOBDD n, Assignment a[1..2^{n.grouping.level}]
     Output: A value in the range of the function represented by n
2    begin
3      return valueTuple[InterpretGrouping(n.grouping, a)];
4    end
5  end
6  SubRoutine InterpretGrouping(g, a)
     Input: Grouping g, Assignment a[1..2^{g.level}]
     Output: An unsigned integer
7    begin
8      if g == ForkGrouping then return 1 + a[1]        // F↦1; T↦2;
9      if g == DontCareGrouping) then return 1                // F,T↦1;
10     if g == NoDistinctionProtoCFLOBDD(g.level) then return 1
        // F,T↦1;
11     Assignment a_A = a[1, 2^{g.level−1}];
12     Assignment a_B = a[2^{g.level−1} + 1, 2^{g.level}];
13     unsigned int i = InterpretGrouping(g.AConnection, a_A);
14     unsigned int k = InterpretGrouping(g.BConnections[i], a_B);
15     return g.BReturnTuples[i](k);
16   end
17 end
```

is reached, the value of the current Boolean variable is consulted (line [8], in the case of a ForkGrouping), or ignored (line [9], in the case of a DontCareGrouping). (Line [10] can be ignored for now; it is an optimization that is discussed in §3.2.5.2.) In lines [13] and [14], Assignment `a` is split in half: the Boolean values in the first half are interpreted during the traversal of `g`'s AConnection (line [13]); the values in the second half are interpreted during the traversal of one of `g`'s BConnections (line [14]), selected according to the value `i` obtained in line [13] from the call on `InterpretGrouping()` with `g`'s AConnection.

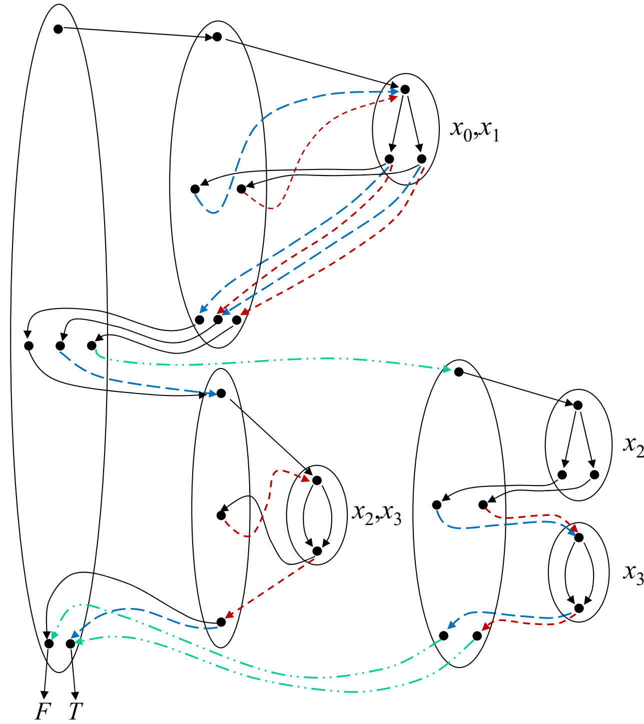


Figure 3.3: CFLOBDD for the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. (For clarity, some of the level-0 groupings have been duplicated.)

3.2.3.5 Multiple Middle Vertices and Exit Vertices

In a Boolean-valued CFLOBDD, the outermost grouping has at most two exit vertices, and these are mapped to $\{F, T\}$. In a multi-terminal CFLOBDD, there can be an arbitrary number of exit vertices, which are mapped to values drawn from some finite set of values V. Fig. 3.1a is a multi-terminal CFLOBDD; the level-1 grouping has two exit vertices that are mapped to 1 and -1.

Groupings that have more than two exit vertices naturally arise in the interior groupings of CFLOBDDs—even in Boolean-valued CFLOBDDs. For instance, a level-i–1 grouping used as an A-connection can more than two exit vertices, in which case the "calling" level-i grouping would have more than two middle vertices. Such multi-terminal groupings can arise in both A-connections and B-connections. Fig. 3.3 shows a Boolean-valued CFLOBDD that contains a level-1 grouping that has three

exit vertices. The grouping is the A-connection of the outermost grouping (at level 2), which thus has three middle vertices.

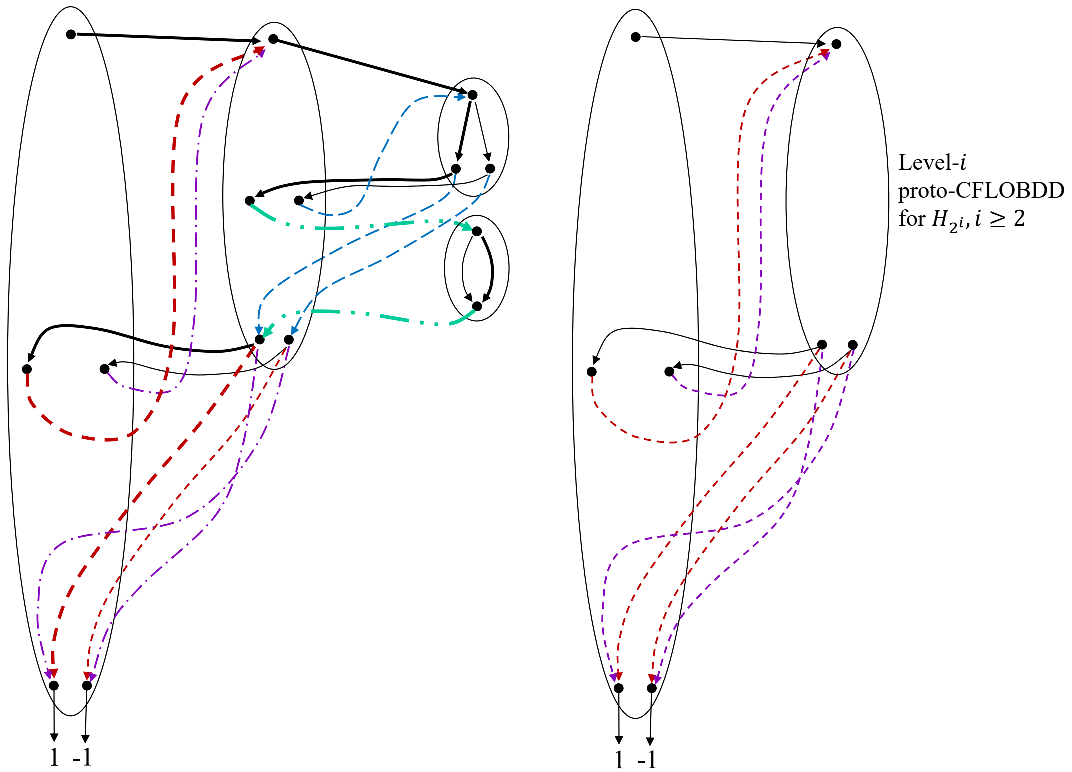
3.2.4 Encoding H_4 and Other Members of $\mathcal{H}$ with a CFLOBDD

Fig. 3.4a shows the CFLOBDD representation of Hadamard matrix H_4 with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. In H_4, the level-1 proto-CFLOBDD is identical to the level-1 proto-CFLOBDD in H_2 (cf. Fig. 3.1a and Fig. 3.4a). Moreover, in H_4 the A-connection call and both B-connection calls are to the level-1 H_2 lookalike.

Consider how Fig. 3.4a encodes $H_4[0,3] = 1$. The value is obtained by evaluating the assignment $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, following the matched path highlighted in bold. The path starts from the level-2 grouping's entry vertex. It goes to the level-1 grouping's entry vertex, where $[x_0 \mapsto F, y_0 \mapsto T]$ is interpreted as for H_2—i.e., the first occurrence of H_2 in "$H_2 \otimes H_2$"—in this case, returning to the leftmost middle vertex of the level-2 grouping. At this point, the path follows the red dashed edge back to the level-1 grouping's entry vertex, where $[x_1 \mapsto F, y_1 \mapsto T]$ is interpreted as for H_2—the second occurrence of H_2 in "$H_2 \otimes H_2$." The path then follows the matching red dashed return edge to the leftmost exit vertex of the level-2 grouping, and reaches terminal value 1.

To create H_4 using H_2, we introduced a level-2 grouping that makes one A-connection and two B-connection "calls" to the level-1 H_2 lookalike, and thus each matched path makes two sequential invocations of H_2. This pattern produces the same effect as the *stacking of plies* in decision trees and BDDs. However, rather than *tripling* the size of the data structure (as with BDDs—see Fig. 2.2e), the ability of CFLOBDDs to reuse parts of a data structure via a "call" means that there is only a *constant-size increase* in going from from H_2 to H_4: one grouping with five vertices and nine edges (one A-connection, two B-connections, and six return edges).

The continuation of this pattern gives an inductive construction of the CFLOBDDs for the other members of $\mathcal{H}$. Given the level-i CFLOBDD for H_{2^i}, $i \geq 2$,



(a) The CFLOBDD representation of H_4 with the interleaved-variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. The matched path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0, 3]$, is shown in bold.

(b) Diagram supporting the inductive argument that, with the interleaved-variable ordering, the members of $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$ can be constructed by successively introducing a new outermost grouping at one greater level. At each step, the same pattern of "calls" is used for the A- and B-connections, and their return edges.

Figure 3.4: Construction of successively larger members of $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$. At level-$(i+1)$, each matched path makes two sequential invocations of the level-i grouping (for H_{2^i}), thereby creating $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$.

$H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$ is created by introducing a new outermost grouping at level $i+1$, again with five vertices and nine edges. (See Fig. 3.4b.) The same pattern of "calls" is used for the A- and B-connections and their return edges: each matched path makes two sequential invocations of the level-i grouping for H_{2^i}. In other words,

Sequential-Invocation Principle. *A Kronecker product $P \otimes Q$ can be represented economically in a CFLOBDD by a grouping at level $i+1$ whose A-connection "calls" the level-i proto-CFLOBDD for P and all of whose B-connection "calls" are to the level-i CFLOBDD for Q.*

3.2.5 Reuse of Groupings and Compression of Boolean Functions

The reason CFLOBDDs can represent certain Boolean functions in a highly compressed fashion is the reuse of groupings that the matched-path and sequential-invocation principles enable.

3.2.5.1 Growth of Number of Paths with Level

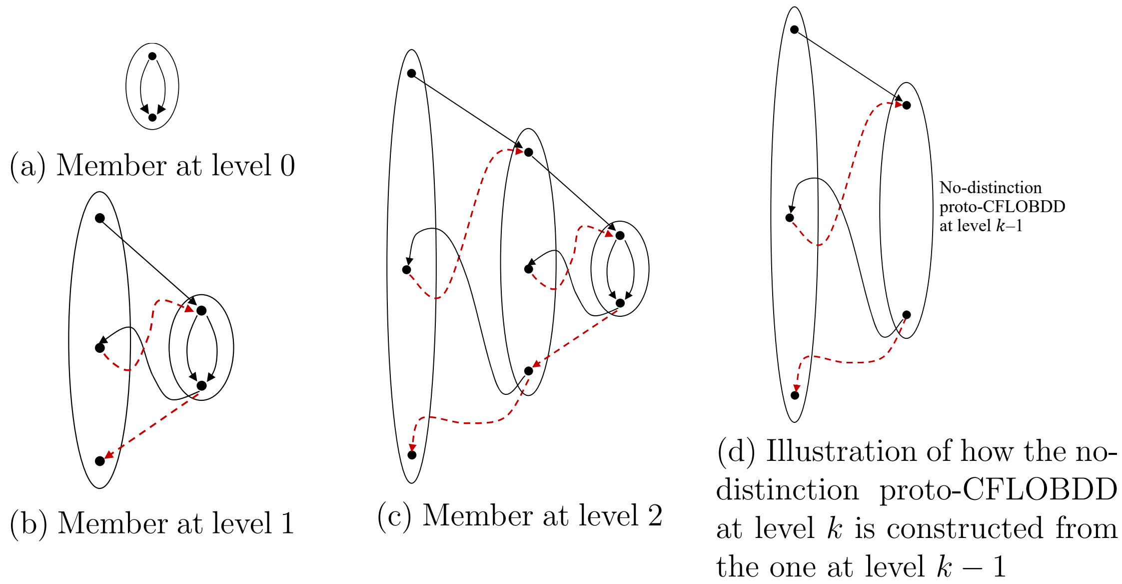
Let $P(i)$ be the number of matched paths in a CFLOBDD at level i. Each level-0 grouping has two paths, so $P(0) = 2$. In a grouping g at level $i \geq 1$, each matched path through the A-connection's level-(i–1) proto-CFLOBDD reaches a middle vertex of g, where it is routed through the level-(i–1) proto-CFLOBDD of the vertex's B-connection. Let $A_j(i-1)$ be the number of matched paths through g's A-connection proto-CFLOBDD to the jth middle vertex of g. Thus, $P(i)$ satisfies the following recurrence equation:

$$P(0) = 2 \qquad\qquad P(i) = \textstyle\sum_j A_j(i-1) \cdot P(i-1). \tag{3.4}$$

The total number of matched paths through g's A-connection proto-CFLOBDD is $P(i-1)$, so $\sum_j A_j(i-1) = P(i-1)$, and hence Eqn. (3.4) can be rewritten as

$$P(i) = P(i-1) \cdot P(i-1), \tag{3.5}$$

which has the solution $P(i) = 2^{2^i}$.



(a) Member at level 0

(b) Member at level 1

(c) Member at level 2

(d) Illustration of how the no-distinction proto-CFLOBDD at level k is constructed from the one at level $k-1$

Figure 3.5: The family of no-distinction proto-CFLOBDDs.

> **Growth in Paths**. *The number of matched paths in a CFLOBDD is* squared *with each increase in level by 1 (Eqn. (3.5)). Consequently, a CFLOBDD at level i has* 2^{2^i} *matched paths.*

3.2.5.2 Best-Case Compression: No-Distinction Proto-CFLOBDDs

Fig. 3.5a, 3.5b, and 3.5c show the first three members of a family of proto-CFLOBDDs that often arise as sub-structures of CFLOBDDs: the single-entry/single-exit proto-CFLOBDDs of levels 0, 1, and 2, respectively. Because every matched path through each of these structures ends up at the unique exit vertex of the highest-level grouping, there is no "decision" to be made during each visit to a level-0 grouping. In essence, as we work our way through such a structure during the interpretation of an assignment, the value assigned to each argument variable makes no difference.

We call this family the *no-distinction proto-CFLOBDDs*. Fig. 3.5d illustrates the structure of a no-distinction proto-CFLOBDD at an arbitrary level $k > 0$, which

continues the pattern that one sees in the level-1 and level-2 structures: the level-k grouping has a single middle vertex, and both its A-connection and its one B-connection are to the no-distinction proto-CFLOBDD for level $k-1$. Moreover, because the no-distinction proto-CFLOBDD at level k shares all but one constant-sized grouping with the no-distinction proto-CFLOBDD at level $k-1$, each additional level costs only a constant amount of additional space. Thus, the no-distinction proto-CFLOBDD at level k is of size $O(k)$, and hence the no-distinction proto-CFLOBDDs exhibit double-exponential compression.

The Boolean-valued CFLOBDD for the constant function $\lambda x_0, x_1, \ldots, x_{2^k-1}.F$ is merely the CFLOBDD in which a value edge connects the (one) exit vertex of the no-distinction proto-CFLOBDD at level k to F. Likewise, in the constant function $\lambda x_0, x_1, \ldots, x_{2^k-1}.T$, the value edge connects the exit vertex of the no-distinction proto-CFLOBDD at level-k to T. Thus, as the number of Boolean variables increases, the best-case growth of CFLOBDDs compares with the growth of decision trees as follows:

Boolean	Number	Decision trees			CFLOBDDs (best case)			
vars.	of paths	height	#nodes	#edges	height[a]	#groupings	#vertices	#edges
1	2	1	3	2	0	1	2	3
2	4	2	7	6	1	2	5	7
4	16	4	31	30	2	3	8	11
8	256	8	511	510	3	4	11	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2^k	2^{2^k}	2^k	$2 \cdot 2^{2^k} - 1$	$2 \cdot 2^{2^k} - 2$	k	$k+1$	$3k+2$	$4k+3$

[a]The height of a CFLOBDD is the level of the outermost grouping.

The best-case CFLOBDD size—whether measured in the number of groupings, vertices, or edges—grows linearly with the level of the outermost grouping, which is *logarithmic* in the number of Boolean variables. In contrast, decision trees grow *exponentially* in the number of Boolean variables. These observations show that Requirement (3) from §3.2.2 is met: in the best case, a decision tree of height 2^k and size 2^{2^k} can be encoded as a level-k CFLOBDD of size k.

Remark Because the family of no-distinction proto-CFLOBDDs is so compact, in designing CFLOBDDs we did not feel the need to mimic the "ply-skipping transformation" of Reduced OBDDs (ROBDDs) Bryant (1986); Brace et al. (1991), in which "don't-care" nodes are removed from the representation. In ROBDDs, in addition to reducing the size of the data structure, the chief benefit of ply-skipping is that operations can skip over levels in portions of the data structure in which no distinctions among variables are made. Essentially the same benefit is obtained by having the algorithms that process CFLOBDDs carry out appropriate special-case processing when no-distinction proto-CFLOBDDs are encountered. Such processing is carried out, for instance, in line [10] of Alg. 1: in `InterpretCFLOBDD()`, when Grouping g is the head of a NoDistinctionProtoCFLOBDD, both g and the entire Assignment a can be ignored because g has only a single exit vertex.

Whereas in the best case, the CFLOBDD for a function f can be double-exponentially smaller than the decision tree for f, ROBDDs are incapable of such a degree of compression. *Quasi-reduced BDDs* are the version of BDDs in which don't-care nodes are *not* removed (i.e., plies are not skipped), and thus all paths from the root to a terminal value have length n, where n is the number of variables. The size of a quasi-reduced BDD is at most a factor of $n+1$ larger than the size of the corresponding ROBDD (Wegener, 2000, Thm. 3.2.3). Thus, although ROBDDs can give better-than-exponential compression compared to decision trees, what one has is not double-exponential compression: at best, it is linear compression of exponential compression. Moreover, in §3.7 we show that the CFLOBDD for a function g can be exponentially smaller than *any* ROBDD for g.

3.2.5.3 Asymptotic Best-Case Compression

Consider a family of functions $F = \{f_j \mid j \geq 0\}$, where the jth member has 2^j Boolean arguments. The following property is a sufficient condition for the sizes of the CFLOBDDs for members of F to grow linearly in the level i, and therefore exhibit double-exponential compression compared to decision trees:

1. There exists a family of functions $G = \{g_j \mid j \geq 0\}$ that grows linearly in the level i.[5]

2. There exists a level m such that, for all levels $i \geq m$,

   (a) the number of vertices in the level-i grouping of f_i is a constant independent of i

   (b) the level-i grouping of f_i makes "procedure calls" only to (i) the level-(i-1) grouping used in the CFLOBDD for $f_{i\text{-}1}$, and (ii) level-(i-1) groupings used in the CFLOBDD for $g_{i\text{-}1}$.[6]

In such a case, the CFLOBDD for each f_i is double-exponentially smaller than the decision tree for f_i—i.e., of size $O(i)$ rather than $O(2^{2^i})$. As shown in Fig. 3.4b, the family of Hadamard matrices $\mathcal{H}$ meets the above conditions.

Moreover, in all cases encountered to date, it is possible to give an explicit algorithm for constructing the ith member of F, where the algorithm runs in time $O(i)$ and uses at most $O(i)$ space.

No information-theoretic limit is being violated here. Not all families of functions can be represented with CFLOBDDs in which each level has a constant number of groupings, each of constant size—and thus, not every function over Boolean-valued arguments can be represented in such a compressed fashion. However, the potential benefit of CFLOBDDs is that, just as with BDDs, there may turn out to be enough regularity in problems that arise in practice that CFLOBDDs stay of manageable size. Moreover, double-exponential compression (or any kind of super-exponential compression) could allow problems to be completed much faster (due to the smaller-sized structures involved), or allow far larger problems to be addressed than has been possible heretofore.

[5] The family of no-distinction proto-CFLOBDDs from Fig. 3.5 is one such family G.

[6] Condition 2b can be generalized so that f_i can "call" the (i-1) groupings used in the CFLOBDDs for some constant number of function families G_1, G_2, ..., G_l that each grow linearly in the level i.

3.3 Canonicalness

In this section, we impose some further structural restrictions on proto-CFLOBDDs and CFLOBDDs that go beyond the ideas illustrated earlier (§3.3.1). We then discuss how to establish that CFLOBDDs are a canonical representation of Boolean functions (§3.3.2 and Appendix §C).

3.3.1 CFLOBDDs Defined, Part II: Additional Structural Invariants

As described in §3.2, the structure of a mock-CFLOBDD consists of different groupings organized into levels, which are connected by edges in a particular fashion. In this section, we describe additional *structural invariants* that are imposed on CFLOBDDs, which go beyond the basic hierarchical structure that is provided by the entry vertex, A-Connection, middle vertices, B-Connections, return edges, and exit vertices of a grouping.

Most of the structural invariants concern the organization of what we call *return tuples* (following the terminology introduced in Fig. 3.2). For a given A-connection edge or B-connection edge c from grouping g_i to g_{i-1}, the return tuple rt_c associated with c consists of the sequence of targets of return edges from g_{i-1} to g_i that correspond to c (listed in the order in which the corresponding exit vertices occur in g_{i-1}). Similarly, the sequence of targets of value edges that emanate from the exit vertices of the highest-level grouping g (listed in the order in which the corresponding exit vertices occur in g) is called the CFLOBDD's *value tuple*.

Return tuples represent mapping functions that map exit vertices at one level to middle vertices or exit vertices at the next greater level. Similarly, value tuples represent mapping functions that map exit vertices of the highest-level grouping to terminal values. In both cases, the ith entry of the tuple indicates the element that the ith exit vertex is mapped to.

Because the middle vertices and exit vertices of a grouping are each arranged in some fixed known order, and hence can be stored in an array, it is often convenient

to assume that each element of a return tuple is simply an index into such an array. For example, in Fig. 3.3,

- The return tuple associated with the 1st B-connection of the upper level-1 grouping is $[1, 2]$.
- The return tuple associated with the 2nd B-connection of the upper level-1 grouping is $[2, 3]$.
- The return tuple associated with the A-connection of the level-2 grouping is $[1, 2, 3]$.
- The value tuple associated with the CFLOBDD is the 2-tuple $[F, T]$.

Rationale

The structural invariants are designed to ensure that—for a given order on the Boolean variables—each Boolean function has a unique, canonical representation as a CFLOBDD. In reading Defn. 3.3 below, it will help to keep in mind that the goal of the invariants is to force there to be a *unique* way to fold a given decision tree into a CFLOBDD that represents the same Boolean function. The decision-tree folding method is discussed in §3.3.2 and Appendix §C, but the main characteristic of the folding method is that it works greedily, left to right. This directional bias shows up in structural invariants 1, 2a, and 2b.

We can now complete the formal definition of a CFLOBDD.

Definition 3.3 Proto-CFLOBDD and CFLOBDD**.** A *proto-CFLOBDD* n is a mock-proto-CFLOBDD (Defns. 3.1 and 3.2) in which every grouping/proto-CFLOBDD in n satisfies the *structural invariants* given below. In particular, let c be an A-connection edge or B-connection edge from grouping g_i to g_{i-1}, with associated return tuple rt_c.

1. If c is an A-connection, then rt_c must map the exit vertices of g_{i-1} one-to-one, and in order, onto the middle vertices of g_i: Given that g_{i-1} has k exit

vertices, there must also be k middle vertices in g_i, and rt_c must be the k-tuple $[1, 2, \ldots, k]$. (That is, when rt_c is considered as a map on indices of exit vertices of g_{i-1}, rt_c is the identity map.)

2. If c is the B-connection edge whose source is middle vertex $j+1$ of g_i and whose target is g_{i-1}, then rt_c must meet two conditions:

   (a) It must map the exit vertices of g_{i-1} one-to-one (but not necessarily onto) the exit vertices of g_i. (That is, there are no repetitions in rt_c.)

   (b) It must "compactly extend" the set of exit vertices in g_i defined by the return tuples for the previous j B-connections: Let rt_{c_1}, rt_{c_2}, ..., rt_{c_j} be the return tuples for the first j B-connection edges out of g_i. Let S be the set of indices of exit vertices of g_i that occur in return tuples rt_{c_1}, rt_{c_2}, ..., rt_{c_j}, and let n be the largest value in S. (That is, n is the index of the rightmost exit vertex of g_i that is a target of any of the return tuples rt_{c_1}, rt_{c_2}, ..., rt_{c_j}.) If S is empty, then let n be 0.

   Now consider rt_c ($= rt_{c_{j+1}}$). Let R be the (not necessarily contiguous) sub-sequence of rt_c whose values are strictly greater than n. Let m be the size of R. Then R must be exactly the sequence $[n+1, n+2, \ldots, n+m]$.

3. While a proto-CFLOBDD may be used as a substructure more than once (i.e., a proto-CFLOBDD may be *pointed to* multiple times), a proto-CFLOBDD never contains two separate *instances* of equal proto-CFLOBDDs.[7]

---

[7] Equality on proto-CFLOBDDs is defined inductively on their hierarchical structure in the obvious manner. Two CFLOBDDs are equal when (i) their proto-CFLOBDDs are equal, and (ii) their value tuples are equal. §3.4.1 discusses how hash-consing Goto (1974) can be used to enforce the invariant that only a single representative CFLOBDD/proto-CFLOBDD exists for each equivalence class of CFLOBDD/proto-CFLOBDD values. However, when we wish to consider the possibility that *multiple* data-structure instances exist that are equal—as we do shortly in §3.3.2—we say that such structures are "isomorphic" or "equal (up to isomorphism)."

To reduce clutter, our diagrams often show multiple instances of the two kinds of level-0 groupings; in fact, a CFLOBDD can contain at most one copy of each.

4. For every pair of B-connections c and c' of grouping g_i, with associated return tuples rt_c and $rt_{c'}$, if c and c' lead to level $i-1$ proto-CFLOBDDs, say p_{i-1} and p'_{i-1}, such that $p_{i-1} = p'_{i-1}$, then the associated return tuples must be different (i.e., $rt_c \neq rt_{c'}$).

A *CFLOBDD* at level k is a mock-CFLOBDD at level k for which

5. The grouping at level k heads a proto-CFLOBDD.

6. The value tuple associated with the grouping at level k maps each exit vertex to a *distinct* value.

Fig. 3.6 illustrates structural invariants 1, 2a, 2b, 3, 4, and 6. In each case, a mock-proto-CFLOBDD that violates one of the structural invariants is shown on the left, and an equivalent proto-CFLOBDD that satisfies the structural invariants is shown on the right.

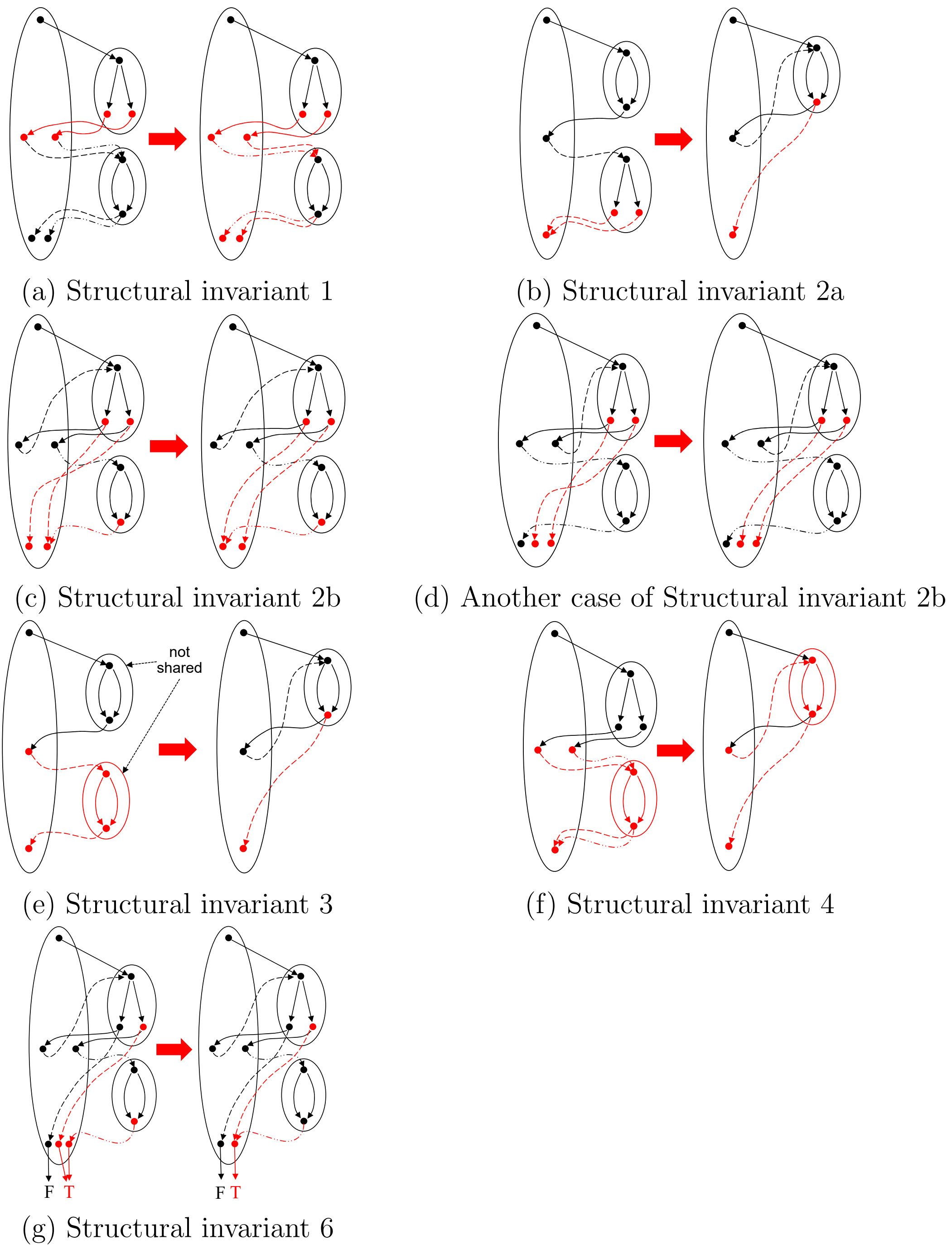
The CFLOBDD from Fig. 3.3 also illustrates the structural invariants.

- The level-1 grouping pointed to by the A-connection of the level-2 grouping has three exit vertices. These are the targets of two return tuples from the uppermost level-0 fork grouping. Note that the blue dashed lines in this proto-CFLOBDD correspond to B-connection 1 and rt_1, whereas the red short-dashed lines correspond to B-connection 2 and rt_2.

  In the case of rt_1, the set S mentioned in structural invariant 2b is empty; therefore, $n = 0$ and rt_1 is constrained by structural invariant 2b to be $[1, 2]$.

  In the case of rt_2, the set S is $\{1, 2\}$, and therefore $n = 2$. The first entry of rt_2, namely 2, falls within the range [1..2]; the second entry of rt_2 lies outside that range and is thus constrained to be 3. Consequently, $rt_2 = [2, 3]$.

  Also in Fig. 3.3, because the level-1 grouping pointed to by the A-connection of the level-2 grouping has three exit vertices, these are constrained by structural



(a) Structural invariant 1

(b) Structural invariant 2a

(c) Structural invariant 2b

(d) Another case of Structural invariant 2b

(e) Structural invariant 3

(f) Structural invariant 4

(g) Structural invariant 6

Figure 3.6: To the left of each arrow, a mock-proto-CFLOBDD that violates the indicated structural invariant; to the right, a corrected proto-CFLOBDD. Invariant violations and their rectifications are shown in red.

invariant 1 to map in order over to the three middle vertices of the level-2 grouping; i.e., the corresponding return tuple is $[1, 2, 3]$.

- The B-connections for the first and second middle vertices of the level-2 grouping are to the same level-1 grouping; however, the two return tuples are different, and thus are consistent with structural invariant 4.

One artifact of the greedy, left-to-right decision-tree folding method used in §3.3.2 and Appendix §C is that matched paths through proto-CFLOBDDs (and hence through CFLOBDDs) have a left-to-right bias in the ordering of paths with respect to Boolean-variable-to-Boolean-value assignments. This bias is captured in the following proposition.

Proposition 3.1 Lexicographic-Order Proposition**.** Let ex_C be the sequence of exit vertices of proto-CFLOBDD C. Let ex_L be the sequence of exit vertices reached by traversing C on each possible Boolean-variable-to-Boolean-value assignment, generated in lexicographic order of assignments. Let s be the subsequence of ex_L that retains just the leftmost occurrences of members of ex_L (arranged in order as they first appear in ex_L). Then $ex_C = s$.

The proof of Prop. 3.1 is provided in Appendix §B.

Earlier in this section, the "Rationale" paragraph motivated the structural invariants as enforcing an implicit "greedy left-to-right folding" of the corresponding decision tree to create the CFLOBDD, and Figure 8 illustrates the structural invariants from a syntactic/operational viewpoint. In contrast, Prop. 3.1 elucidates a semantic consequence of the structural invariants.[8]

Example 3.1. Prop. 3.1 can be illustrated using Fig. 3.3. If we use numbers to identify exit vertices, ex_C for any grouping g is the sequence $[1..g.\text{numberOfExits}]$. In

[8]§3.3.2 gives a high-level overview of the proof that CFLOBDDs are a canonical representation of Boolean functions. In the proof of canonicity in §C, Prop. 3.1 is used in the proof of Prop. C.1, which establishes property (3) from §3.3.2.

the upper level-1 grouping in Fig. 3.3, ex_L is $[1, 2, 2, 3]$, so s is $[1, 2, 3]$. In the level-1 grouping at the lower right, ex_L is $[1, 1, 2, 2]$, so s is $[1, 2]$. In the level-2 grouping, ex_L is $[1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2]$, so s is $[1, 2]$.

3.3.2 Canonicity of CFLOBDDs

CFLOBDDs are a canonical representation of functions over Boolean arguments, i.e., each decision tree with 2^{2^k} leaves is represented by exactly one isomorphism class of level-k CFLOBDDs. (The notion of isomorphism of CFLOBDDs was introduced in footnote 7.)

Theorem 3.2 Canonicity**.** If C_1 and C_2 are level-k CFLOBDDs for the same Boolean function over 2^k Boolean variables, and C_1 and C_2 use the same variable ordering, then C_1 and C_2 are isomorphic.

To prove this theorem, we make use of Obs. 2.1, and argue not in terms of Boolean functions but in terms of *representations* of Boolean functions—specifically, we relate two kinds of Boolean-function representations

- the decision tree T_B for a Boolean function B, using some fixed, but otherwise unspecified, variable ordering Ord, and
- the CFLOBDD for B, again using variable ordering Ord.

By Obs. 2.1, we use T_B as a stand-in for B, thereby avoiding having to talk about B itself. In particular, we must establish that three properties hold:

1. Every level-k CFLOBDD represents a decision tree with 2^{2^k} leaves.
2. Every decision tree with 2^{2^k} leaves is represented by some level-k CFLOBDD.
3. No decision tree with 2^{2^k} leaves is represented by more than one level-k CFLOBDD (up to isomorphism).

The proof that CFLOBDDs are a canonical representation of Boolean functions is in Appendix §C.

We already showed that Obligation 1 is satisfied in §3.2.3.4.

Obligation 2 is established by showing that there is a recursive procedure for constructing a level-k CFLOBDD from an arbitrary decision tree with 2^{2^k} leaves (i.e., of height 2^k)—see Construction 1 in Appendix §C. In essence, the construction shows how such a decision tree can be folded together to form a CFLOBDD that represents the same Boolean function. The construction ensures that the structural invariants are obeyed.

Obligation 3 is established by showing that (i) unfolding a CFLOBDD C into a decision tree T and then (ii) folding T back to a CFLOBDD yields a CFLOBDD that is isomorphic to C. In particular, the folding-back step applies the same algorithm we use to establish Obligation 2, namely, Construction 1 from Appendix §C. Construction 1 is a *deterministic algorithm*, and thus the proof establishes that T can only be mapped to a CFLOBDD C' that is isomorphic to C. (See Prop. C.1.)

Note that Obligation 1 and 2 are exactly Requirements (1) and (2) from §3.2.2, respectively. Moreover, Obligations 1–3 together show that Requirement (4) from §3.2.2 is met.

3.4 Pragmatics

The structure of the groupings in a CFLOBDD is acyclic: a level-k grouping has calls exclusively to groupings at level k-1; conversely, a given grouping at level k-1 can be called from multiple groupings, but only ones at level k. This property allows CFLOBDDs to be implemented in a functional style without side-effects. Moreover, because groupings are acyclic, storage can be managed via smart-pointer-based reference counting.

The remainder of this section discusses pragmatics—namely, how some of the

standard techniques for working with a functional data structure apply to CFLOBDDs. All three of the techniques discussed contribute to an implementation being able to satisfy Requirement (5) that operations on a CFLOBDD run in time polynomial in the sizes of (i) the input CFLOBDDs, or (ii) the input CFLOBDDs and the output CFLOBDD.

3.4.1 Hash-Consing of Groupings and CFLOBDDs to Create Unique Representatives

Hash-consing Goto (1974) enforces the invariant that only a single representative exists for each value constructed from some datatype. Hash-consing should not be confused with canonicity (§3.3.2 and Appendix §C). Canonicity is a semantic property: if two CFLOBDDs C_1 and C_2 represent the same function, then C_1 and C_2 are isomorphic. Hash-consing concerns concrete memory representations: for a given data-structure construction pattern, only a single representative exists in memory, no matter how many times that value arises in a computation.

However, because canonicity holds for CFLOBDDs, an implementation that uses hash-consing[9] satisfies an even stronger form of equivalence. In particular, Thm. 3.2 can be restated to read "... then C_1 and C_2 are identical."

Because the operations that construct `Grouping`s and `CFLOBDD`s involve a certain amount of processing before the object being constructed is finally complete, we will assume that two operations, named `RepresentativeGrouping` and `RepresentativeCFLOBDD`, are available for explicitly maintaining the tables of representative `Grouping`s and `CFLOBDD`s, respectively. For instance, a call `RepresentativeGrouping(g)` checks to see whether a representative for `g` is already in the table of representative `Grouping`s; if there is such a representative, say `h`, then `g` is discarded and `h` is returned as the result; if there is no such represen-

[9]It can also be useful to use hash-consing for the objects of classes `ReturnTuple`, `PairTuple`, and `ValueTuple`.

tative, then `g` is installed in the table and returned as the result. The operations `RepresentativeForkGrouping` and `RepresentativeDontCareGrouping` return the unique representatives of types `ForkGrouping` and `DontCareGrouping`, respectively.

Operations discussed in §3.6 that create `InternalGrouping`s, such as `PairProduct` (Alg. 12) and `Reduce` (Alg. 14), have the following form:

```
Operation() {
  ...
  InternalGrouping g = new InternalGrouping(k);
  ...
  // Operations to fill in the members of g, including g.AConnection and the
  // elements of array g.BConnections, with level k-1 Groupings
  ...
  return RepresentativeGrouping(g);
}
```

The operation `NoDistinctionProtoCFLOBDD` (Alg. 3), which constructs the members of the family of no-distinction proto-CFLOBDDs depicted in Fig. 3.5, also has this form.

`RepresentativeCFLOBDD` is similar to `RepresentativeGrouping`, but in addition to a `Grouping` argument, it also has a value-tuple argument. The operation `ConstantCFLOBDD` (Alg. 2) illustrates the use of `RepresentativeCFLOBDD`: `ConstantCFLOBDD(k,v)` returns a hash-consed CFLOBDD that represents a constant function of the form $\lambda x_0, x_1, \ldots, x_{2^i-1}.v$.

In our implementation, we maintain the invariant that the Groupings that appear in the hash-consing tables are the heads of fully-fledged proto-CFLOBDDs, not mock-proto-CFLOBDDs—i.e., structural invariants (1)–(4) of Defn. 3.3 hold. When a proto-CFLOBDD p is associated with terminal values to create a CFLOBDD c, it is necessary to ensure that structural invariant (6) holds. In particular, if there are any duplicate terminal values, a "reduction" step is applied (see Alg. 14 of §3.6.3), which may cause smaller versions of some of the groupings in p to be constructed. The

original groupings would be collected if their reference counts go to 0. However, there is never any issue of the hash-cons tables being polluted by mock-proto-CFLOBDDs that violate the proto-CFLOBDD structural invariants.

3.4.2 Equality Testing for CFLOBDDs and proto-CFLOBDDs

As discussed in §3.4.1, the combined effect of hash-consing and canonicity is that an implementation can maintain the invariant that, at any given time, there is a unique concrete memory representation of a given Boolean function. Consequently, it is possible to test in unit time—by comparing two pointers—whether two variables of type `CFLOBDD` represent the same Boolean function. This property is important in user-level applications in which various kinds of data are implemented using class `CFLOBDD`. For example, in applications structured as fixed-point-finding loops, this property provides a unit-cost test of whether the fixed-point has been reached.

Again, because of the use of hash-consing, it is also possible to test whether two variables of type `Grouping` are equal via a single pointer comparision. Because each grouping is always the highest-level grouping of some proto-CFLOBDD, the equality test on `Grouping`s is really a test of whether two proto-CFLOBDDs are equal. The property of being able to test two proto-CFLOBDDs for equality quickly is important because proto-CFLOBDD equality tests are used during the various operations on CFLOBDDs to maintain the structural invariants from Defn. 3.3.

Finally, the ability to test two proto-CFLOBDDs for equality quickly also allows some functions—typically near the beginning of the function—to identify important special-case values of parameters, which can lead to faster performance. For instance, in Alg. 1, line [10], we saw how testing whether the argument g is a NoDistinctionProtoCFLOBDD allows further recursive calls to `InterpretGrouping()` to be short-circuited.

3.4.3 Function Caching

A function cache (or *memo function* Michie (1967)) for a function F is an associative-lookup table—typically a hash table—of pairs of the form $[x, F(x)]$, keyed on the value of x. The table is consulted each time F is applied to some argument, and updated after a return value is computed for a never-before-seen argument. The technique saves the cost of re-performing the computation of F for an argument on which F has previously been called, at the expense of performing a lookup on F's argument at the beginning of each call. Our implementation of CFLOBDDs uses function caching for a number of the operations described in the remainder of the paper, such as `PairProduct` (Alg. 12) and `Reduce` (Alg. 14). To reduce clutter in the pseudo-code that we give, we elide the lines for querying and updating the cache. The full statement of such a function would have the following form:

```
F(x) {
    if cache_F(x) ≠ NULL return cache_F(x);
    ...
    cache_F(x) = retVal; // Update the cache with the return value
    return retVal;
}
```

Function caching involves hashing, and it is necessary to perform equality tests to resolve hash collisions. Thus, the ability to test two proto-CFLOBDDs for equality in unit time (§3.4.2) also improves the performance of function caching.

3.5 A Denotational Semantics

In §3.2.3.4, we gave an operational semantic definition of the function that a CFLOBDD represents. In this section, we give denotational semantics, which defines the function that a CFLOBDD denotes. In particular, this semantics associates the ith element of a CFLOBDD's valueTuple with the set of Assignments (i.e., language of bit-strings) that map to the ith element. That is, a CFLOBDD n at level k denotes

a function

$$[\![n]\!] : [1..|n.valueTuple|] \to \mathcal{P}(\{0,1\}^{2^k}).$$

Moreover, the $|n.valueTuple|$ range values form a partition of $\{0,1\}^{2^k}$. (That is, the range values are pairwise disjoint, and $\bigcup_{i=1}^{|valueTuple|} [\![n]\!][i] = \{0,1\}^{2^k}$.)

The definition of $[\![n]\!]$ is given in terms of a recursive definition of the semantics of Groupings (really proto-CFLOBDDs). Consider a Grouping g at level l with m exit vertices. Suppose that $g.AConnection$ has p exits, and $g.BConnections[i]$ has k_i exit vertices. The return map $g.AReturnTuple$ is always a 1-1 map, and hence it will not play an explicit role in defining $[\![g]\!]$, but $g.BReturnTuples[i]$—the return edges from $g.BConnections[i]$'s exit vertices to g's exit vertices—does play a role. For convenience, we define $[\![g]\!]$ to be a vector, of dimension $1 \times m$. The m entries of the vector form a partition of $\{0,1\}^{2^l}$. In particular, the vectors for a ForkGrouping g_{Fork} and a DontCareGrouping g_{DC} are

$$[\![g_{Fork}]\!] \stackrel{\text{def}}{=} [\{0\},\{1\}] \qquad\qquad [\![g_{DC}]\!] \stackrel{\text{def}}{=} [\{0,1\}].$$

The semantics of a Grouping g at level l is defined recursively in terms of g's A- and B-connection Groupings at level $l-1$. To give such a definition, we need to define the meaning of $g.BReturnTuples[i]$, the return edges from the exit vertices of a Grouping's i^{th} B-connection. We define $[\![g.BReturnTuples[i]]\!]$ to be a "permutation matrix" of size $k_i \times m$. Each entry of the matrix is either $\emptyset$ or $\{\epsilon\}$, where ϵ denotes the empty string, with the properties that (i) every row must have exactly one occurrence of $\{\epsilon\}$, and (ii) every column must have at most one occurrence of $\{\epsilon\}$. For example, if $g.BReturnTuples[i]$ maps $g.BConnections[i]$'s 3 exit vertices into g's 5 exit vertices by $[1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 3]$, then

$$[\![g.BReturnTuples[i]]\!] = \begin{bmatrix} \emptyset & \{\epsilon\} & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{\epsilon\} & \emptyset \\ \emptyset & \emptyset & \{\epsilon\} & \emptyset & \emptyset \end{bmatrix}_{3\times 5}$$

We now define $[\![g]\!]$ recursively, where the subscripts denote the dimensions of the vector or matrix, and the two matrix-multiplication primitives are language concatenation and language union:

$$[\![g]\!]_{1\times m} = \begin{cases} [\{0\},\{1\}]_{1\times 2} & \text{if g} = \mathit{ForkGrouping} \\ [\{0,1\}]_{1\times 1} & \text{if g} = \mathit{DontCareGrouping} \\ [\![g.AConnection]\!]_{1\times p} \times \begin{bmatrix} \vdots \\ [\![g.BConnections[i]]\!]_{1\times k_i} \times \\ \quad [\![g.BReturnTuples[i]]\!]_{k_i\times m} \\ \vdots \end{bmatrix}_{\substack{p\times m \\ i\in\{1..p\}}} & \text{otherwise} \end{cases}$$

Finally, for a CFLOBDD n, $[\![n]\!]$ is defined as follows:

$$[\![n]\!]_{1\times|n.valueTuple|} \stackrel{\text{def}}{=} [\![n.grouping]\!]_{1\times|n.grouping.numberOfExits|}$$

Example 3.2. For the five proto-CFLOBDDs depicted in Fig. 3.7, the vectors of languages are as follows (read top-to-bottom by level):

level 2	level 1	level 0
$\begin{bmatrix} \{0000,0001,0010,0100,0101,0110, \\ 1000,1001,1010\}, \\ \{1100,1101,1110,1111,0011,0111,1011\} \end{bmatrix}$	$[\{00,01,10\},\{11\}]$	$[\{0\},\{1\}]$
	$[\{00,01,10,11\}]$	$[\{0,1\}]$

3.6 Algorithms on CFLOBDDs

In this section, we describe operations to construct or combine CFLOBDDs. To aid the reader, Tab. 3.1 lists the fourteen main operations on CFLOBDDs, together with references to where the algorithm for each operation is presented (and where it is discussed), along with each operation's asymptotic running time and the asymptotic

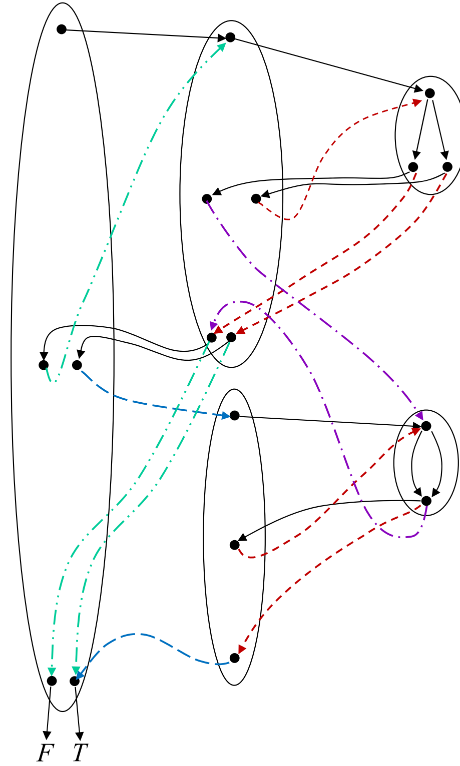


Figure 3.7: CFLOBDD representation of the function $\lambda w, x, y, z.(w \wedge x) \vee (y \wedge z)$, with variable ordering $\langle w, x, y, z \rangle$

running time of the analogous BDD operation. Readers familiar with BDDs will find that the algorithms for operations on CFLOBDDs are somewhat more complicated than their BDD counterparts, mainly due to the need to maintain the CFLOBDD structural invariants (Defn. 3.3).

3.6.1 Primitive CFLOBDD-Creation Operations

3.6.1.1 Constant Functions

The CFLOBDD-creation operation `ConstantCFLOBDD`, given as Alg. 2, produces the family of CFLOBDDs that represent functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1}.v$, where v is some constant value. `ConstantCFLOBDD`(k, v) uses as a subroutine `NoDistinctionProtoCFLOBDD` (Alg. 3), which constructs the no-distinction proto-CFLOBDD for a given level k (see also Fig. 3.5). `ConstantCFLOBDD`

Operation	Type Signature	Description	Time Complexity	
			CFLOBDD	BDD
Equal (§3.4.2)	CFLOBDD × CFLOBDD → Boolean	Checks if two CFLOBDDs are equal	$\mathcal{O}(1)$	$\mathcal{O}(1)$
ConstantCFLOBDD (Alg. 2, §3.6.1.1)	Int(k) × Value(v) → CFLOBDD	Creates a CFLOBDD for a constant function $\lambda x_0 \ldots x_{2^k-1}.v$	$\mathcal{O}(k)$	$\mathcal{O}(2^k)$
FalseCFLOBDD (Alg. 4, §3.6.1.1)	Int (k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.F$	$\mathcal{O}(k)$	$\mathcal{O}(2^k)$
TrueCFLOBDD (Alg. 5, §3.6.1.1)	Int (k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.T$	$\mathcal{O}(k)$	$\mathcal{O}(2^k)$
NoDistinctionProtoCFLOBDD (Alg. 3, §3.6.1.1)	Int (k) → Proto-CFLOBDD	Creates a NoDistinctionProtoCFLOBDD for 2^k variables	$\mathcal{O}(k)$	N/A
ProjectionCFLOBDD (Algs. 6 and 7, §3.6.1.2)	Int(k) × Int(i) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.x_i$	$\mathcal{O}(k)$	$\mathcal{O}(2^k)$
FlipValueTupleCFLOBDD (Alg. 8, §3.6.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are flipped	$\mathcal{O}(1)$	$\mathcal{O}(\lvert c\rvert_B)$
ComplementCFLOBDD (Alg. 8, §3.6.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are complemented	$\mathcal{O}(1)$	$\mathcal{O}(\lvert c\rvert_B)$
ScalarMultiplyCFLOBDD (Alg. 9, §3.6.2.2)	CFLOBDD (c) × Value (v) → CFLOBDD	Performs $c' = c * v$	$\mathcal{O}(\lvert c\rvert \times \lvert c'\rvert)$	$\mathcal{O}(\lvert c\rvert_B)$
BinaryApplyAndReduce (Alg. 11, §3.6.3)	CFLOBDD (c_1) × CFLOBDD (c_2) × Operation op → CFLOBDD	Performs $c' = c_1 \, op \, c_2$	$\mathcal{O}(\lvert c_1\rvert \times \lvert c_2\rvert \times \lvert c'\rvert)$	$\mathcal{O}(\lvert c_1\rvert_B \times \lvert c_2\rvert_B)$
PathCounting (Alg. 23, §3.6.8.1)	CFLOBDD(c) → CFLOBDD	Computes the number of paths to every exit vertex of every grouping	$\mathcal{O}(\lvert c\rvert)$	$\mathcal{O}(\lvert c\rvert_B)$ (See §3.9.1.2.)
Sampling (Algs. 24 and 25, §3.6.8.2)	CFLOBDD(c) → String	Samples a path from c	$\mathcal{O}(\max(vars, \lvert c\rvert))$	$\mathcal{O}(\max(vars, \lvert c\rvert_B))$ (See §3.9.1.2.)
KroneckerProduct (App.§G & Alg. 18, §3.6.5)	CFLOBDD(c_1) × CFLOBDD(c_2) → CFLOBDD	Performs $c' = c_1 \otimes c_2$	$\mathcal{O}((\lvert c_1\rvert + \lvert c_2\rvert + c_1.\#\text{exits} \times c_2.\#\text{exits}) \times \lvert c'\rvert)$	$\mathcal{O}(\lvert c_1\rvert_B)$
MatrixMultiply (Algs. 20, 21, and 22 , §3.6.7)	CFLOBDD(c_1) × CFLOBDD(c_2) → CFLOBDD	Performs $c' = c_1 \times c_2$ for matrices of size $N \times N$	$\mathcal{O}(N^3)$, plus the time for a final call to `Reduce`	$\mathcal{O}(N^3)$

Table 3.1: List of operations on CFLOBDDs; *vars* denotes the number of Boolean variables ($= 2^k$, where k is the number of levels of the CFLOBDD). The size measure $|\cdot|$ counts the number of groupings, vertices, and edges—with no double-counting of shared groupings due to hash-consing. In the column for the time complexities of BDD operations, an occurrence of c refers to a BDD argument of the operation, and $|c|_B$ denotes the size of BDD c (the number of nodes and edges). For quasi-reduced BDDs, the time to construct the analog of NoDistinctionProtoCFLOBDD is $\mathcal{O}(2^k)$. Note that the complexity of MatrixMultiply is in terms of the sizes of matrices represented by c_1 and c_2 and not the sizes of c_1 and c_2.

Algorithm 2: ConstantCFLOBDD

Input: int k, Value v
Output: CFLOBDD representation of a function with 2^k variables and constant value v

```
1 begin
2 |  return RepresentativeCFLOBDD(NoDistinctionProtoCFLOBDD(k),
  |    [v]);
3 end
```

Algorithm 3: NoDistinctionProtoCFLOBDD

Input: int k
Output: Proto-CFLOBDD representation of a function with 2^k variables

```
1  begin
2  |  if k == 0 then
3  |  |  return RepresentativeDontCareGrouping;
4  |  end
5  |  InternalGrouping g = new InternalGrouping(k);
6  |  g.AConnection = NoDistinctionProtoCFLOBDD(k-1);
7  |  g.AReturnTuple = [1];
8  |  g.numberOfBConnections = 1;
9  |  g.BConnections[1] = g.AConnection;
10 |  g.BReturnTuples[1] = [1];
11 |  g.numberOfExits = 1;
12 |  return RepresentativeGrouping(g);
13 end
```

Algorithm 4: FalseCFLOBDD

Input: int k
Output: CFLOBDD representation of a function with 2^k variables and constant value F

```
1 begin
2 |  return ConstantCFLOBDD(k, F);
3 end
```

can be used to construct CFLOBDDs for the constant functions $\lambda x_0, x_1, \ldots, x_{2^k-1}.F$ (Alg. 4) and $\lambda x_0, x_1, \ldots, x_{2^k-1}.T$ (Alg. 5). `ConstantCFLOBDD`(k, v) runs in time $O(k)$ and uses at most $O(k)$ space.

Algorithm 5: TrueCFLOBDD

```
    Input: int k
    Output: CFLOBDD representation of a function with 2^k variables and
            constant value T
1 begin
2 |  return ConstantCFLOBDD(k, T);
3 end
```

Algorithm 6: ProjectionProtoCFLOBDD

```
1 Algorithm ProjectionCFLOBDD(k, i)
  |  Input: int k (level), int i (index)
  |  Output: CFLOBDD representing function λx_0, x_1, ..., x_{2^k-1}.x_i
2 |  begin
3 |  |  assert(0 <= i < 2**k);
4 |  |  return
  |  |   RepresentativeCFLOBDD(ProjectionProtoCFLOBDD(k,i),
  |  |   [F,T]);
5 |  end
6 end
```

3.6.1.2 Projection Functions

A second family of CFLOBDD-creation operations produces the Boolean-valued *(single-variable) projection functions* of the form $\lambda x_0, x_1, \ldots, x_{2^k-1}.x_i$, where i ranges from 0 to $2^k - 1$. Fig. 3.8 illustrates the structure of the CFLOBDDs that represent these functions. Algs. 6 and 7 give pseudo-code for `ProjectionCFLOBDD`(k, i), which constructs the ith such function. `ProjectionCFLOBDD`(k, i) runs in time $O(k)$ and uses at most $O(k)$ space.

3.6.2 Unary Operations on CFLOBDDs

This section discusses how to perform certain unary operations on CFLOBDDs:

Algorithm 7: ProjectionProtoCFLOBDD (cont.)

```
1  SubRoutine ProjectionProtoCFLOBDD(k, i)
     Input: int k (level), int i (index)
     Output: Grouping g representing function λx_0, x_1, ..., x_{2^k-1}.x_i
2    begin
3      if k == 0 then                                // i must also be 0
4        return RepresentativeForkGrouping;
5      else
6        InternalGrouping g = new InternalGrouping(k);
7        if i < 2**(k-1) then        // i falls in AConnection range
8          g.AConnection = ProjectionProtoCFLOBDD(k-1,i);
9          g.AReturnTuple = [1,2];
10         g.numBConnections = 2;
11         g.BConnection[1] = NoDistinctionProtoCFLOBDD(k-1);
12         g.BReturnTuples[2] = [1];
13         g.BConnections[2] = g.BConnection[1];
14         g.BReturnTuples[2] = [2];
15         g.numberOfExits = 2;
16       else                        // i falls in BConnection range
17         g.AConnection = NoDistinctionProtoCFLOBDD(k-1);
18         g.AReturnTuple = [1];
19         g.numBConnections = 1;
20         i = i - 2**(k-1);            // Remove high-order bit for
            recursive call
21         g.BConnections[1] = ProjectionProtoCFLOBDD(k-1,i);
22         g.BReturnTuples[1] = [1,2];
23         g.numberOfExits = 2;
24       end
25       return RepresentativeGrouping(g);
26     end
27   end
28 end
```

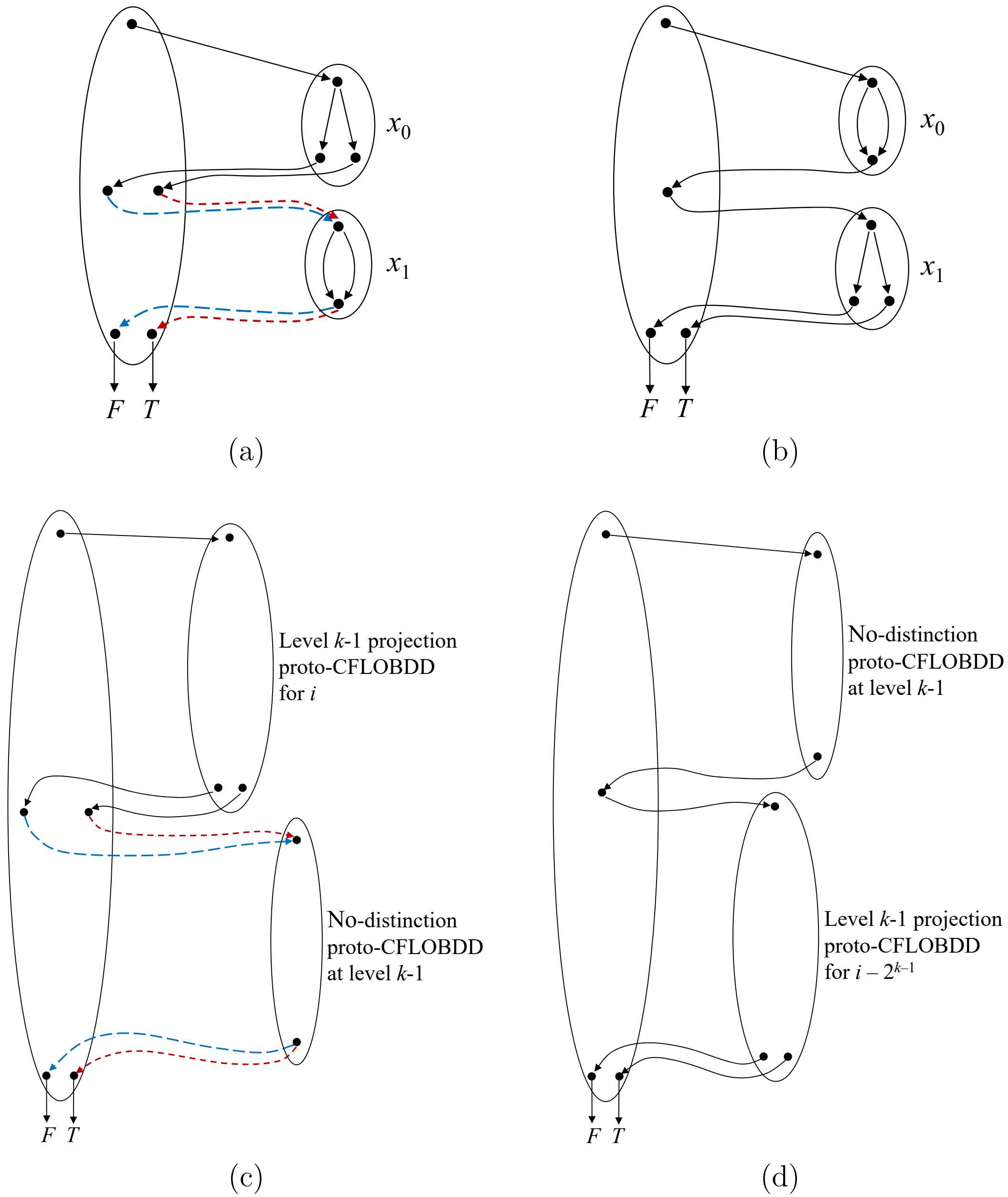


Figure 3.8: (a) CFLOBDD for $\lambda x_0 x_1 . x_0$; (b) CFLOBDD for $\lambda x_0 x_1 . x_1$; (c) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1} . x_i$, when $0 \leq i < 2^{k-1}$; (d) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1} . x_i$, when $2^{k-1} \leq i < 2^k$.

Algorithm 8: ComplementCFLOBDD

```
1  Algorithm FlipValueTupleCFLOBDD(c)
     Input: CFLOBDD c
     Output: CFLOBDD c′ such that the output values are flipped
2    begin
3      assert(|c.valueTuple| == 2);
4      return RepresentativeCFLOBDD(c.grouping, [c.valueTuple[2],
         c.valueTuple[1]]);
5    end
6  end
7  Algorithm ComplementCFLOBDD(c)
     Input: CFLOBDD c
     Output: CFLOBDD c′ such that the output values are complemented
8    begin
9      if c == FalseCFLOBDD(c.grouping.level) then
10       return TrueCFLOBDD(c.grouping.level);
11     end
12     if c == TrueCFLOBDD(c.grouping.level) then
13       return FalseCFLOBDD(c.grouping.level);
14     end
15     return FlipValueTupleCFLOBDD(c);
16   end
17 end
```

Algorithm 9: ScalarMultiplyCFLOBDD

```
   Input: CFLOBDD c, Value v
   Output: CFLOBDD c′ = c ∗ v
1  begin
     // Multiply CFLOBDD c by the CFLOBDD for the constant function
          λx_0, x_1, ..., x_{2^k−1}.v
2    return BinaryApplyAndReduce(c, ConstantCFLOBDD(c.level, v),
       (op)Times);                                    // (See §3.6.3)
3  end
```

3.6.2.1 FlipValueTuple Function

The function `FlipValueTupleCFLOBDD` applies in the special situation in which a CFLOBDD maps Boolean-variable-to-Boolean-value assignments to just two possible values; `FlipValueTupleCFLOBDD` flips the two values in the CFL-

OBDD's `valueTuple` field and returns the resulting CFLOBDD. In the case of Boolean-valued CFLOBDDs, this operation can be used to implement the operation `ComplementCFLOBDD`, which forms the Boolean complement of its argument, in an efficient manner. The pseudocode for these functions is given in Alg. 8. `FlipValueTupleCFLOBDD` and `ComplementCFLOBDD` are constant-time operations.

3.6.2.2 Scalar Multiplication

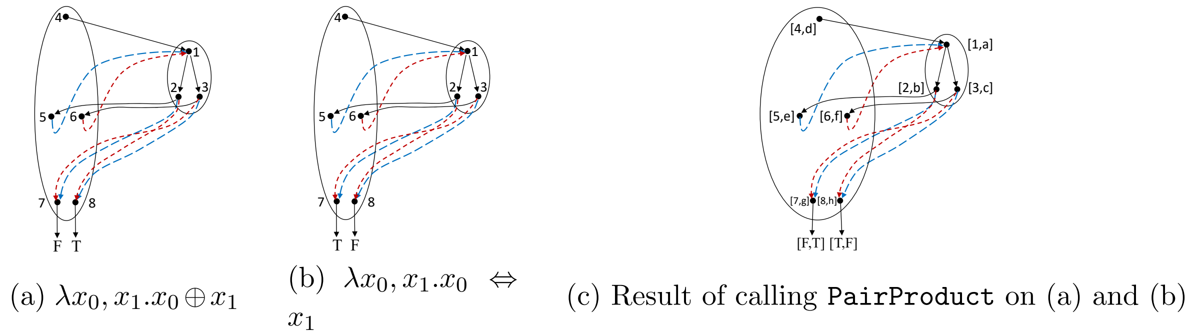
Function `ScalarMultiplyCFLOBDD` of Alg. 9 applies to any CFLOBDD that maps Boolean-variable-to-Boolean-value assignments to values on which multiplication by a scalar value of type `Value` is defined. `ScalarMultiplyCFLOBDD` constructs a CFLOBDD for the constant function $\lambda x_0, x_1, \ldots, x_{2^k-1}.v$, which is multiplied by CFLOBDD c using `BinaryApplyAndReduce`—the generic operation for binary CFLOBDD operations (discussed in §3.6.3)—with the multiplication operator Times passed as the third argument.

3.6.3 Binary Operations on CFLOBDDs

This section presents an algorithm for performing binary operations on CFLOBDDs. The algorithm is parameterized in terms of a binary operation `op` that is to be applied pointwise to the range values of two CFLOBDDs. That is, given the CFLOBDDs for two functions n_1 and n_2 and binary operation `op`, the goal of the algorithm is to create the CFLOBDD for $n_1 \,\texttt{op}\, n_2$ where, for each assignment a, $(n_1 \,\texttt{op}\, n_2)(a) = n_1(a) \,\texttt{op}\, n_2(a)$. Operation `op` could be $+, -, *, /$, etc., or—if the functions are Boolean-valued—$\vee, \wedge, \oplus$, etc. As with BDDs, such operations on CFLOBDDs can be implemented via a two-step process[10]

1. perform a product construction

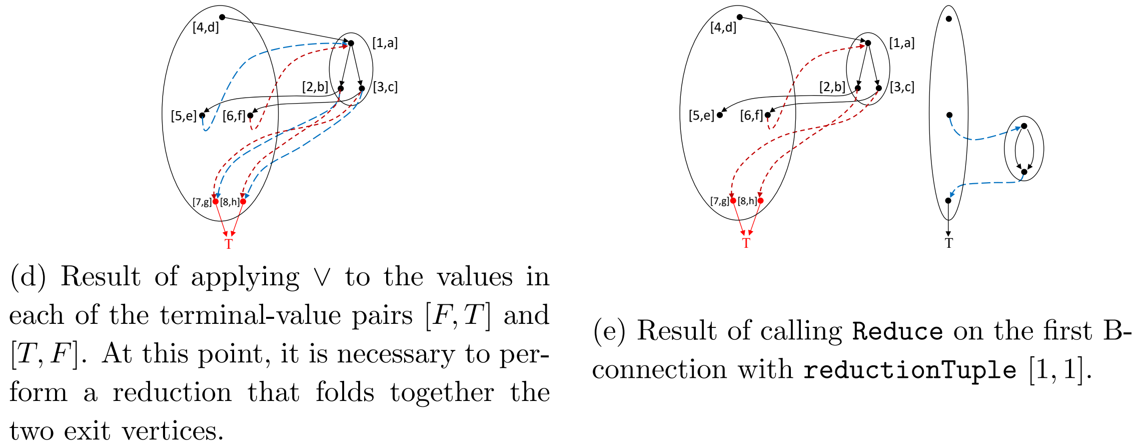
[10]The two-step process is conceptual for BDDs: the two steps can be combined in an implementation (e.g., see (Filliâtre and Conchon, 2006, §3.3), (Boyer and Hunt Jr, 2006, §7)). For CFLOBDDs, it does not appear possible to combine the two steps, at least not easily. For more details, see the Remark just after Ex. 3.3.



(a) $\lambda x_0, x_1.x_0 \oplus x_1$

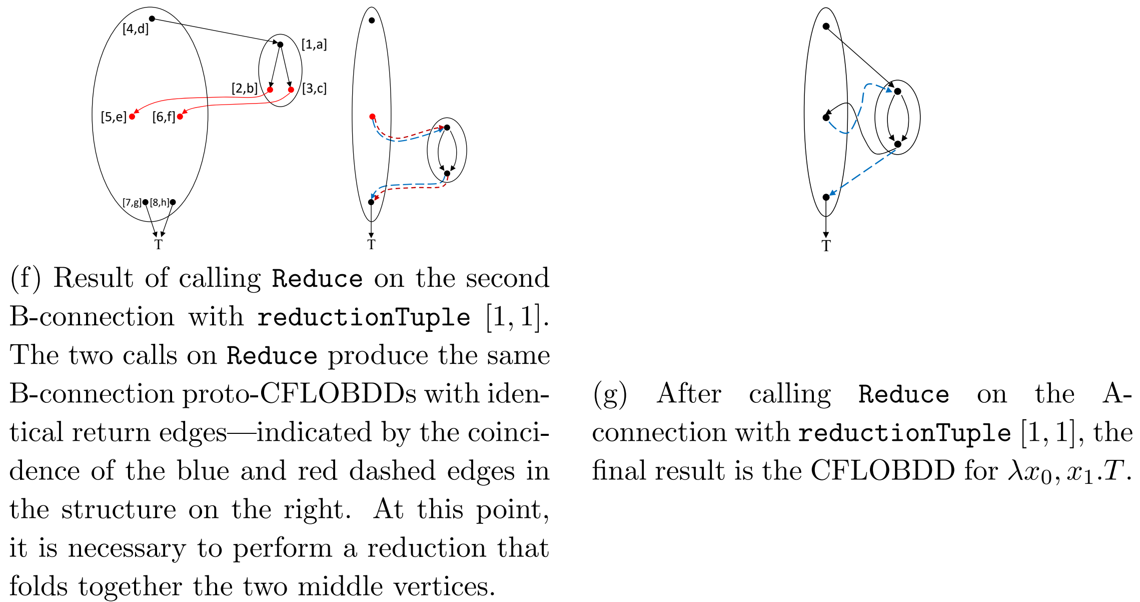
(b) $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$

(c) Result of calling `PairProduct` on (a) and (b)



(d) Result of applying $\vee$ to the values in each of the terminal-value pairs $[F, T]$ and $[T, F]$. At this point, it is necessary to perform a reduction that folds together the two exit vertices.

(e) Result of calling `Reduce` on the first B-connection with `reductionTuple` $[1, 1]$.



(f) Result of calling `Reduce` on the second B-connection with `reductionTuple` $[1, 1]$. The two calls on `Reduce` produce the same B-connection proto-CFLOBDDs with identical return edges—indicated by the coincidence of the blue and red dashed edges in the structure on the right. At this point, it is necessary to perform a reduction that folds together the two middle vertices.

(g) After calling `Reduce` on the A-connection with `reductionTuple` $[1, 1]$, the final result is the CFLOBDD for $\lambda x_0, x_1.T$.

Figure 3.9: Illustrating how $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$ results in $\lambda x_0, x_1.T$.

2. perform a reduction step on the result of step one.

Just as there can be multiple occurrences of a given node in a BDD, there can be multiple occurrences of a given grouping in a CFLOBDD. To avoid a blow-up in costs, binary operations need to avoid making repeated calls on a given pair of groupings $g_1 \in n_1$ and $g_2 \in n_2$. Assuming that the hash-table lookup and insertion methods used for hash-consing (§3.4.1) and function caching (§3.4.3) run in (expected) unit-cost time, the time to perform the product construction is asymptotically bounded by the product of the sizes of the two argument CFLOBDDs—i.e., $O(|n_1| \times |n_2|)$.[11] §K shows that the time for the reduction step is $O(|n_1| \times |n_2| \times |n'|)$, where n' denotes the CFLOBDD that is the result of $n_1 \, \texttt{op} \, n_2$. Consequently, binary operations satisfy Requirement (5); i.e., they run in (expected) time that is polynomial in the sizes of the input and output CFLOBDDs.

Fig. 3.9 illustrates this method by showing how the CFLOBDD for $\lambda x_0, x_1.T$ is obtained as the result of a Boolean-$\vee$: $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$. Fig. 3.9c shows the result of the product construction (`PairProduct`, Alg. 12). Fig. 3.9d, e, f, and g illustrate some of the steps of the reduction algorithm (`Reduce`, Alg. 14). Fig. 3.9 is discussed in more detail in Ex. 3.3.

The algorithms involved are given as Algs. 10–15. (In Algs. 11 and 12, we assume that the `CFLOBDD` or `Grouping` arguments are objects whose highest-level groupings are all at the same level.)

- The operation `BinaryApplyAndReduce`, given as Alg. 11, starts with a call on `PairProduct` (line [2]). `PairProduct`, given as Algs. 12 and 13, performs a recursive traversal of the two `Grouping` arguments, `g1` and `g2`, to create a proto-CFLOBDD that represents a kind of cross product. `PairProduct` returns `g`,

[11] More precisely, let $n{\uparrow}gr[k]$ denote the set of groupings at level $k{\in}[0..l]$ in CFLOBDD n. The time to construct the product of n_1 and n_2 is asymptotically bounded by $\sum_{k=0}^{l} \sum \left\{ |g_1| \times |g_2| \mid g_1 {\in} n_1{\uparrow}gr[k] \text{ and } g_2 {\in} n_2{\uparrow}gr[k] \right\}$.

Algorithm 10: CollapseClassesLeftmost

```
   Input: Tuple equivClasses
   Output: Tuple×Tuple [projectedClasses, renumberedClasses]
1  begin
     // Project the tuple equivClasses, preserving
        left-to-right order, retaining the leftmost instance of
        each class
2    Tuple projectedClasses = [equivClasses(i) : i ∈ [1..|equivClasses|] | i =
       min{j ∈ [1..|equivClasses|] | equivClasses(j) = equivClasses(i)}];
     // Create tuple in which classes in equivClasses are
        renumbered according to their ordinal position in
        projectedClasses
3    Map orderOfProjectedClasses = {[x,i]: i ∈ [1..|projectedClasses|] | x =
       projectedClasses(i)};
4    Tuple renumberedClasses = [orderOfProjectedClasses(v) : v ∈
       equivClasses];
5    return [projectedClasses, renumberedClasses];
6  end
```

Algorithm 11: BinaryApplyAndReduce

```
   Input: CFLOBDDs n1, n2 and Operation op
   Output: CFLOBDD n = n1 op n2
1  begin
     // Perform cross product
2    Grouping×PairTuple [g,pt] = PairProduct(n1.grouping,n2.grouping);
     // Create tuple of ``leaf'' values
3    ValueTuple deducedValueTuple = [
       op(n1.valueTuple[i1],n2.valueTuple[i2]) : [i1,i2] ∈ pt ];
     // Collapse duplicate leaf values, folding to the left
4    Tuple×Tuple [inducedValueTuple,inducedReductionTuple] =
       CollapseClassesLeftmost(deducedValueTuple) ;
     // Perform corresponding reduction on g, folding g's exit
        vertices w.r.t. inducedReductionTuple
5    Grouping g' = Reduce(g, inducedReductionTuple) ;
6    CFLOBDD n = RepresentativeCFLOBDD(g', inducedValueTuple) ;
7    return n;
8  end
```

Algorithm 12: PairProduct

```
  Input: Groupings g1, g2
  Output: Grouping g: product of g1 and g2; PairTuple ptAns: tuple of
          pairs of exit vertices
1 begin
2   if g1 and g2 are both no-distinction proto-CFLOBDDs then return
      [ g1, [[1,1]] ];
3   if g1 is a no-distinction proto-CFLOBDD then return [ g2, [[1,k] : k
      ∈ [1..g2.numberOfExits]] ];
4   if g2 is a no-distinction proto-CFLOBDD then return [ g1, [[k,1] : k
      ∈ [1..g1.numberOfExits]] ];
5   if g1 and g2 are both fork groupings then return [ g1, [[1,1],[2,2]] ];
    // Pair the A-connections
6   Grouping×PairTuple [gA,ptA] = PairProduct(g1.AConnection,
      g2.AConnection);
7   InternalGrouping g = new InternalGrouping(g1.level);
8   g.AConnection = gA ;
9   g.AReturnTuple = [1..|ptA|]; // Represents the middle vertices
10  g.numberOfBConnections = |ptA| ;
    // Continued in Alg. 13
11 end
```

the proto-CFLOBDD formed in this way, as well as `pt`, a descriptor of the exit vertices of `g` in terms of pairs of exit vertices of the highest-level groupings of `g1` and `g2`. (See Alg. 12, lines [2]–[5] and Alg. 13, lines [20]–[38].)

From the semantic perspective, each exit vertex e_1 of `g1` represents a (non-empty) set A_1 of variable-to-Boolean-value assignments that lead to e_1 along a matched path in `g1`; similarly, each exit vertex e_2 of `g2` represents a (non-empty) set of variable-to-Boolean-value assignments A_2 that lead to e_2 along a matched path in `g2`. If `pt`, the descriptor of `g`'s exit vertices returned by `PairProduct`, indicates that exit vertex e of `g` corresponds to $[e_1, e_2]$, then e represents the (non-empty) set of assignments $A_1 \cap A_2$.

Function caching (§3.4.3) is performed for `PairProduct`. Consequently, for a given invocation of `BinaryApplyAndReduce` on CFLOBDDs n_1 and n_2, for

Algorithm 13: PairProduct (cont.)

```
19
      // Pair the B-connections, but only for pairs in ptA
      // Descriptor of pairings of exit vertices
20    Tuple ptAns = [];
      // Create a B-connection for each middle vertex
21    for j ← 1 to |ptA| do
22        Grouping×PairTuple [gB,ptB] =
           PairProduct(g1.BConnections[ptA(j)(1)],
           g2.BConnections[ptA(j)(2)]);
23        g.BConnections[j] = gB ;
          // Now create g.BReturnTuples[j], and augment ptAns as
             necessary
24        g.BReturnTuples[j] = [] ;
25        for i ← 1 to |ptB| do
26            c1 = g1.BReturnTuples[ptA(j)(1)](ptB(i)(1));      // an exit
               vertex of g1
27            c2 = g2.BReturnTuples[ptA(j)(2)](ptB(i)(2)) ;     // an exit
               vertex of g2
28            if [c1,c2] ∈ ptAns then     // Not a new exit vertex of g
29                index = the k such that ptAns(k) == [c1,c2] ;
30                g.BReturnTuples[j] = g.BReturnTuples[j] || index ;
31            else                 // Identified a new exit vertex of g
32                g.numberOfExits = g.numberOfExits + 1 ;
33                g.BReturnTuples[j] = g.BReturnTuples[j] ||
                   g.numberOfExits ;
34                ptAns = ptAns || [c1,c2] ;
35            end
36        end
37    end
38    return [RepresentativeGrouping(g), ptAns];
39 end
```

Algorithm 14: Reduce

```
   Input: Grouping g, ReductionTuple reductionTuple
   Output: Grouping g' that is "reduced"
 1 begin
       // Test whether any reduction actually needs to be carried
          out
 2     if reductionTuple == [1..|reductionTuple|] then
 3         return g;
 4     end
       // If only one exit vertex, then collapse to
          no-distinction proto-CFLOBDD
 5     if |{x : x ∈ reductionTuple}| == 1 then
 6         return NoDistinctionProtoCFLOBDD(g.level);
 7     end
 8     InternalGrouping g' = new InternalGrouping(g.level);
 9     g'.numberOfExits = |{ x : x ∈ reductionTuple }|;
10     Tuple reductionTupleA = [];
11     for i ← 1 to g.numberOfBConnections do
12         Tuple deducedReturnClasses = [reductionTuple(v) : v ∈
             g.BReturnTuples[i]];
13         Tuple×Tuple [inducedReturnTuple, inducedReductionTuple] =
             CollapseClassesLeftmost(deducedReturnClasses);
14         Grouping h = Reduce(g.BConnection[i], inducedReturnTuple);
15         int position = InsertBConnection(g', h, inducedReturnTuple);
16         reductionTupleA = reductionTupleA || position;
17     end
18     Tuple×Tuple [inducedReturnTuple, inducedReductionTuple] =
         CollapseClassesLeftmost(reductionTupleA);
19     Grouping h' = Reduce(g.AConnection, inducedReductionTuple);
20     g'.AConnection = h';
21     g'.AReturnTuple = inducedReturnTuple;
22     return RepresentativeGrouping(g');
23 end
```

Algorithm 15: InsertBConnection

Input: InternalGrouping g, Grouping h, ReturnTuple returnTuple
Output: int – Insert (h, ReturnTuple) as the next B-connection of g, if they are a new combination; otherwise return the index of the existing occurrence of (h, ReturnTuple)

```
1 begin
2   if there exists i ∈ [1..g.numberOfBConnections] such that
      g.BConnection[i] == h && g.BReturnTuples[i] == returnTuple then
      return i;
3   g.numberOfBConnections = g.numberOfBConnections + 1;
4   g.BConnections[g.numberOfBConnections] = h;
5   g.BReturnTuples[g.numberOfBConnections] = returnTuple;
6   return g.numberOfBConnections;
7 end
```

each level k, the number of calls on `PairProduct` for level k is bounded by the product of the numbers of level-k groupings in n_1 and n_2. Moreover, for each call on `PairProduct`(g_1, g_2), the number of exit vertices in grouping g is bounded by the product of the numbers of exit vertices in g_1 and g_2 (see line [34]). Similarly, the number of middle vertices in g is bounded by the product of the numbers of middle vertices in g_1 and g_2 (see line [10]). Thus, the size of `g` is bounded by the product of the sizes of g_1 and g_2. Consequently, the cost of the call on `PairProduct` in line [2] of Alg. 11 is bounded by the sum over $k \in [0..l]$ of the products of the sizes of the level-k groupings in n_1 and n_2, and hence polynomial in the sizes of n_1 and n_2 (see footnote 11).

Lines [2]–[4] of `PairProduct` perform special-case processing when either argument to `PairProduct` is a NoDistinctionProtoCFLOBDD. At level 0, these checks—along with line [5]—implement the base case of `PairProduct`. However, at levels greater than 0, they allow `PairProduct` to return immediately, without making any recursive calls to traverse g_1 or g_2, potentially saving considerable work.

- `BinaryApplyAndReduce` then uses `pt`, together with `op` and the value tuples

from `CFLOBDD`s `n1` and `n2`, to create the tuple `deducedValueTuple` of leaf values that should be associated with the exit vertices (see Alg. 11, line [3]]).

However, `deducedValueTuple` is a *tentative* value tuple for the constructed `CFLOBDD`; because of Structural Invariant 6, this tuple needs to be collapsed if it contains duplicate values.

- `BinaryApplyAndReduce` obtains two tuples, `inducedValueTuple` and `inducedReductionTuple`, which describe the collapsing of duplicate leaf values, by calling the subroutine `CollapseClassesLeftmost` (Alg. 10):
  - Tuple `inducedValueTuple` serves as the final value tuple for the `CFLOBDD` constructed by `BinaryApplyAndReduce`. In `inducedValueTuple`, the leftmost occurrence of a value in `deducedValueTuple` is retained as the representative for that equivalence class of values. For example, if `deducedValueTuple` is $[2, 2, 1, 1, 4, 1, 1]$, then `inducedValueTuple` is $[2, 1, 4]$.

    The use of leftward folding is dictated by Structural Invariant 2b.
  - Tuple `inducedReductionTuple` describes the collapsing of duplicate values that took place in creating `inducedValueTuple` from `deducedValueTuple`: `inducedReductionTuple` is the same length as `deducedValueTuple`, but each entry `inducedReductionTuple(i)` gives the ordinal position of `deducedValueTuple(i)` in `inducedValueTuple`. For example, if `deducedValueTuple` is $[2, 2, 1, 1, 4, 1, 1]$ (and thus `inducedValueTuple` is $[2, 1, 4]$), then `inducedReductionTuple` is $[1, 1, 2, 2, 3, 2, 2]$—meaning that positions 1 and 2 in `deducedValueTuple` were folded to position 1 in `inducedValueTuple`, positions 3, 4, 6, and 7 were folded to position 2 in `inducedValueTuple`, and position 5 was folded to position 3 in `inducedValueTuple`.

(See Alg. 11, line [4], as well as Alg. 10.)

- Finally, `BinaryApplyAndReduce` performs a corresponding reduction on `Grouping g`, by calling the subroutine `Reduce`, which creates a new `Grouping` in which `g`'s exit vertices are folded together with respect to tuple `inducedReductionTuple` (Alg. 11, line [5]).

  Procedure `Reduce`, given as Alg. 14, recursively traverses `Grouping g`, working in the backwards direction, first processing each of `g`'s B-connections in turn, and then processing `g`'s A-connection. In both cases, the processing is similar to the (leftward) collapsing of duplicate leaf values that is carried out by `BinaryApplyAndReduce`:

  - In the case of each B-connection, rather than collapsing with respect to a tuple of duplicate final values, `Reduce`'s actions are controlled by its second argument, `reductionTuple`, which clients of `Reduce`—namely, `BinaryApplyAndReduce` and `Reduce` itself—use to inform `Reduce` how `g`'s exit vertices are to be folded together. For instance, the value of `reductionTuple` could be $[1, 1, 2, 2, 3, 2, 2]$—meaning that exit vertices 1 and 2 are to be folded together to form exit vertex 1, exit vertices 3, 4, 6, and 7 are to be folded together to form exit vertex 2, and exit vertex 5 by itself is to form exit vertex 3.

    In Alg. 14, line [12], the value of `reductionTuple` is used to create a tuple that indicates the equivalence classes of targets of return edges for the B-connection under consideration (in terms of the new exit vertices in the `Grouping` that will be created to replace `g`).

    Then, by calling the subroutine `CollapseClassesLeftmost`, `Reduce` obtains two tuples, `inducedReturnTuple` and `inducedReductionTuple`, that describe the collapsing that needs to be carried out on the exit vertices of the B-connection under consideration (Alg. 14, line [13]).

Tuple `inducedReductionTuple` is used to make a recursive call on `Reduce` to process the B-connection; `inducedReturnTuple` is used as the return tuple for the `Grouping` returned from that call. Note how the call on `InsertBConnection` (Alg. 15) in line [15] of `Reduce` enforces structural invariant 4 of Defn. 3.3.[12]

– As the B-connections are processed, `Reduce` uses the `position` information returned from `InsertBConnection` to build up the tuple `reductionTupleA` (Alg. 14, line [16]). This tuple indicates how to reduce the A-connection of `g`.

– Finally, via processing similar to what was done for each B-connection, two tuples are obtained that describe the collapsing that needs to be carried out on the exit vertices of the A-connection, and an additional call on `Reduce` is carried out. (See Alg. 14, lines [18]–[21].)

Function caching (§3.4.3) is performed for `Reduce`, with respect to both arguments `g` and `reductionTuple`.

§K shows that the time for a call to Reduce(n, rt) with output CFLOBDD n' is asymptotically bounded by $O(|n| \times |n'|)$. Because the time for `PairProduct` to perform the product construction of two CFLOBDDs n_1 and n_2 is asymptotically bounded by the product of their sizes (i.e., $O(|n_1| \times |n_2|)$), the overall time to perform BinaryApplyAndReduce(n_1, n_2) is $O(|n_1| \times |n_2| \times |n'|)$, which is polynomial in the sizes of the input and output CFLOBDDs.

Recall that a call on `RepresentativeGrouping(g)` may have the side effect of installing `g` into the table of memoized `Grouping`s. We do not wish for this table to ever be polluted by non-well-formed proto-CFLOBDDs. Thus, there is a subtle point as to why the grouping `g` constructed during a call on `PairProduct` meets structural

[12]In our implementation, `InsertBConnection` performs a left-to-right search of `g.BConnection` and `g.BReturnTuples`, but it could be implemented as an (expected) unit-time operation using a hashed dictionary, keyed on (Grouping, ReturnTuple) pairs.

invariant 4—and hence why it is permissible to call `RepresentativeGrouping(g)` in line [38] of Alg. 13. We give this proof in Appendix §D.

Lastly, in the case of Boolean-valued CFLOBDDs, there are 16 possible binary operations, corresponding to the 16 possible two-argument truth tables (2×2 matrices with Boolean entries). All 16 possible binary operations are special cases of `BinaryApplyAndReduce`; these can be performed by passing `BinaryApplyAndReduce` an appropriate value for argument `op` (i.e., some 2×2 Boolean matrix).

Example 3.3. Fig. 3.9 illustrates how the CFLOBDD for $\lambda x_0, x_1.T$ is created from the "or" ($\vee$) of the CFLOBDDs for $\lambda x_0, x_1.x_0 \oplus x_1$ and $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$. Fig. 3.9c is the result of calling `PairProduct` on the CFLOBDDs for $\lambda x_0, x_1.x_0 \oplus x_1$ and $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$. After $\vee$ is applied to the values in each of the terminal-value pairs $[F, T]$ and $[T, F]$, we obtain a mock-CFLOBDD that has two exit vertices associated with terminal value T. To restore the structural invariants and create a CFLOBDD, the two exit vertices must be folded together, and a reduction performed on each of the two B-connections. In each case, `Reduce` is called with `reductionTuple` $[1, 1]$. Because these reductions result in the same B-connection proto-CFLOBDDs with identical return edges (Fig. 3.9e and Fig. 3.9f), which would be discovered by `InsertBConnection` (Alg. 15), it is necessary to fold together the two middle vertices and perform a reduction on the A-connection: `Reduce` is called with `reductionTuple` $[1, 1]$, This step produces the CFLOBDD for $\lambda x_0, x_1.T$ (Fig. 3.9g).

For another example that illustrates `Reduce`, see Ex. K.1.

Remark For BDDs, the two-step process of "pair-product-followed-by-reduction" need only be conceptual. Binary operations on BDDs can be implemented during a single recursive pass by performing the appropriate value-reduction operation on terminal values, and then, as the recursion unwinds, having the BDD-node constructor perform hash-consing (suppressing the construction of don't-care nodes) so that

non-reduced structures are never created ((Filliâtre and Conchon, 2006, §3.3), (Boyer and Hunt Jr, 2006, §7)).

Such an approach does not seem to be possible with CFLOBDDs because reduction is not obtained as a side-effect of hash-consing. The flow of control in `Reduce` (Alg. 14) follows the sequence of elements of a matched path backwards. `Reduce` makes recursive calls for the B-connection proto-CFLOBDDs and then a recursive call for the A-connection proto-CFLOBDD (rather than working bottom-up from level-0 groupings to level-k groupings, which would be the analogue of the bottom-up construction performed with BDDs.) Consequently, our CFLOBDD implementation maintains the weaker invariant that the `Grouping`s that appear in the hash-consing tables are the heads of fully-fledged proto-CFLOBDDs, not mock-proto-CFLOBDDs—i.e., structural invariants (1)–(4) of Defn. 3.3 hold. While such `Grouping`s may have to be reduced later, there is never any issue of the hash-cons tables being polluted by mock-proto-CFLOBDDs that violate the proto-CFLOBDD structural invariants.

Some unary operations on CFLOBDDs may also need to apply `Reduce`. For example, if the terminal values of a CFLOBDD are numeric values, the unary function that squares all terminal values could initially result in a mock-CFLOBDD that has duplicate terminal values. `Reduce`, with an appropriate `ReductionTuple`, would be then applied to create the corresponding CFLOBDD.

In a manner similar to the binary operations on CFLOBDDs, we can perform ternary operations on CFLOBDDs. More details about how to perform these operations can be found in Appendix §E.1. Other operations, such as restriction ($f|_{x_i=v}$) and existential quantification ($\exists x_i.f$) can also be performed on a CFLOBDD; the corresponding algorithms can be found in Appendices §E.2 and §E.3, respectively.

3.6.4 Representing Matrices and Vectors using CFLOBDDs

Matrices and vectors are important data structures used in quantum simulation (§3.9.2.2). In this subsection and following subsections, we discuss how to

represent Boolean or non-Boolean matrices and vectors using CFLOBDDs, and how to perform operations on such representations of matrices and vectors.

3.6.4.1 Matrix Representation

We represent square matrices using CFLOBDDs by having the Boolean variables correspond to bit positions in the row and column indices. That is, suppose that M is a $2^n \times 2^n$ matrix; M is represented using a CFLOBDD over $2n$ Boolean variables $\{x_0, x_1, \ldots, x_{n-1}\} \cup \{y_0, y_1, \ldots, y_{n-1}\}$, where the variables $\{x_0, x_1, \ldots, x_{n-1}\}$ represent the successive bits of x—the first index into M—and the variables $\{y_0, y_1, \ldots, y_{n-1}\}$ represent the successive bits of y—the second index into M, with $\log n + 1$ levels.[13] The indices of elements of matrices represented in this way start at 0; for example, the upper-left corner element of a matrix M is $M(0,0)$. When $n = 2$, $M(0,0)$ corresponds to the value associated with the assignment $[x_0 \mapsto 0, x_1 \mapsto 0, y_0 \mapsto 0, y_1 \mapsto 0]$.

It is often convenient to use either the *interleaved* ordering i.e., the order of the Boolean variables is chosen to be $x_0, y_0, x_1, y_1, \ldots, x_{n-1}, y_{n-1}$—or the *reverse interleaved* ordering—i.e., the order is $y_{n-1}, x_{n-1}, y_{n-2}, y_{n-2}, \ldots, y_0, x_0$.

One nice property of the interleaved-variable ordering is that, as we work through each pair of variables in an assignment, the matrix elements that remain "in play" represent a sub-block of the full matrix. For instance, suppose that we have a Boolean matrix whose entries are defined by the function $\lambda x_0 y_0 x_1 y_1 . (x_0 \wedge y_0) \vee (x_1 \wedge y_1)$, as shown below:

		0	0	1	1	y_0
		0	1	0	1	y_1
0	0	F	F	F	F	
0	1	F	T	F	T	
1	0	F	F	T	T	
1	1	F	T	T	T	
x_0	x_1					

[13] Matrices of other sizes, including non-square matrices, can be represented by embedding them within a larger square matrix. For matrices with ¿2 dimensions, there would be a set of Boolean variables for the index-bits of each dimension.

If we were to evaluate the 16 possible assignments in lexicographic order, i.e., in the order

$$\begin{array}{c}
[x_0 \mapsto 0, y_0 \mapsto 0, x_1 \mapsto 0, y_1 \mapsto 0], \\
[x_0 \mapsto 0, y_0 \mapsto 0, x_1 \mapsto 0, y_1 \mapsto 1], \\
[x_0 \mapsto 0, y_0 \mapsto 0, x_1 \mapsto 1, y_1 \mapsto 0], \\
[x_0 \mapsto 0, y_0 \mapsto 0, x_1 \mapsto 1, y_1 \mapsto 1], \\
[x_0 \mapsto 0, y_0 \mapsto 1, x_1 \mapsto 0, y_1 \mapsto 0], \\
\vdots \\
[x_0 \mapsto 1, y_0 \mapsto 1, x_1 \mapsto 1, y_1 \mapsto 0], \\
[x_0 \mapsto 1, y_0 \mapsto 1, x_1 \mapsto 1, y_1 \mapsto 1]
\end{array}$$

then we would step through the array elements in the order shown below:

		0	0	1	1	y_0
		0	1	0	1	y_1
0	0	1	2	5	6	
0	1	3	4	7	8	
1	0	9	10	13	14	
1	1	11	12	15	16	
x_0	x_1					

If the first two elements of an assignment are $[x_0 \mapsto 0, y_0 \mapsto 1]$, the elements still in play are the ones in the positions labeled 5, 6, 7, and 8 in the upper-right quadrant.

There is an important, non-standard consequence of using a CFLOBDD to represent a matrix that very likely is not apparent from the discussion above, having to do with the sizes of subproblems in a divide-and-conquer algorithm. In fact, the same issue arises in designing a divide-and-conquer algorithm over any data structure represented via a CFLOBDD, as illustrated in Fig. 3.10. Suppose that a CFLOBDD C represents a decision tree T_C that has $\log_2 P$ variables, and thus P leaves. The A-connection proto-CFLOBDD accounts for half the variables, namely, $\frac{\log_2 P}{2}$, and the B-connection proto-CFLOBDDs account for the remaining half. The natural way to divide C in a divide-and-conquer algorithm is at the middle vertices of the outermost grouping: process the A-connection proto-CFLOBDD, and then the B-connection proto-CFLOBDDs (or vice versa). In T_C, this division corresponds to

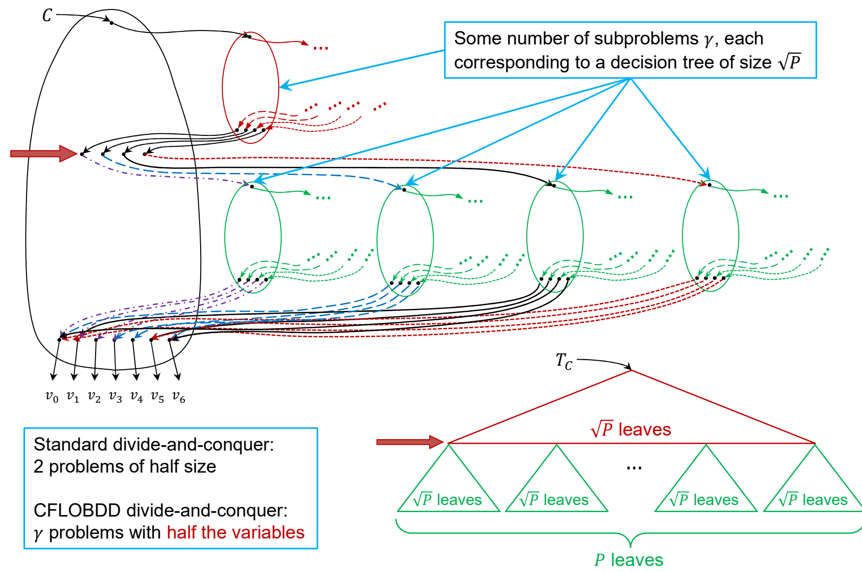


Figure 3.10: Why a $\sqrt{P} \times \sqrt{P}$-decomposition is the natural problem decomposition for divide-and-conquer algorithms on structures represented as CFLOBDDs.

the tree partitioning shown in the lower-right corner of Fig. 3.10: C's A-connection proto-CFLOBDD corresponds to the red tree rooted at the apex of T_C (which has $\sqrt{P}$ leaves); C's B-connection proto-CFLOBDDs correspond to the $\sqrt{P}$ green trees in the bottom half of T_C (each of which has $\sqrt{P}$ leaves). In contrast with standard divide-and-conquer algorithms, which often divide a problem into two subproblems of half size, this approach divides the original problem into $\sqrt{P}+1$ subproblems, each of size $O(\sqrt{P})$. With CFLOBDDs, in contrast to decision trees, there is the potential for subproblems to be shared among the A-connection and B-connections, so one ends up with some number of subproblems γ ($= 1 +$ the number of middle vertices), each of size $O(\sqrt{P})$.

Whereas with a decision tree it would be easy to take the conventional approach of dividing a problem into two problems of half *size*—using the left child and right child of the apex, in essence "peeling off" the topmost ply, such a division is not convenient for CFLOBDDs because the decision variable for the topmost ply is associated with the level-0 grouping found by following the A-connection of the

A-connection of the ..., etc. For CFLOBDDs, the natural structure of a divide-and-conquer algorithm lies with the A-connection proto-CFLOBDD and the set of B-connection proto-CFLOBDDs—a division based on dividing *the number of variables* in half.

In certain cases, including matrix multiplication (§3.6.7), the $\gamma \times \sqrt{P}$-decomposition structure forced us to rethink how to perform various algorithms.

Let us now consider how such a decomposition works for an $N \times N$ matrix M, assuming the interleaved-variable ordering, where $N = 2^n$. Thus, n is the number of bits in a row-index (respectively, column-index); there are $2n$ Boolean variables in total; and $P = N^2$. M would be decomposed into $\sqrt{P} = \sqrt{N^2} = N$ sub-matrices, each of size $\sqrt{N} \times \sqrt{N}$. At top level, the A-connection of the CFLOBDD for M captures commonalities in the $\sqrt{N} \times \sqrt{N}$ block structure of M, and the B-connections represent the blocks: sub-matrices of M of size $\sqrt{N} \times \sqrt{N}$.

For instance, when a level-3 CFLOBDD is used to represent a matrix, there are $2n = 8 = 2^3$ index variables—i.e., $n = 4$ variables for each dimension—so the matrix is of size 16×16. Its natural constituents are level-2 proto-CFLOBDDs, which each have $2^2 = 4$ index variables. Thus, there are 2 A-connection variables for each dimension of the block structure, and 2 B-connection variables for each dimension of the sub-matrix for a block. Consequently, a matrix of size 16×16 is decomposed into 16 ($= 4 \times 4 = \sqrt{16} \times \sqrt{16}$) blocks, each of size $4 \times 4 = \sqrt{16} \times \sqrt{16}$, as indicated below:

				0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	y_0
				0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	y_1
				0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	y_2
				0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	y_3
0	0	0	0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
0	0	0	1	F	T	F	T	F	F	F	F	F	F	F	F	F	F	F	F	
0	0	1	0	F	F	T	T	F	F	F	F	F	F	F	F	F	F	F	F	
0	0	1	1	F	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F	
0	1	0	0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
0	1	0	1	F	F	F	F	F	T	F	T	F	F	F	F	F	F	F	F	
0	1	1	0	F	F	F	F	F	F	T	T	F	F	F	F	F	F	F	F	
0	1	1	1	F	F	F	F	F	T	T	T	F	F	F	F	F	F	F	F	
1	0	0	0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
1	0	0	1	F	F	F	F	F	F	F	F	F	T	F	T	F	F	F	F	
1	0	1	0	F	F	F	F	F	F	F	F	F	F	T	T	F	F	F	F	
1	0	1	1	F	F	F	F	F	F	F	F	F	T	T	T	F	F	F	F	
1	1	0	0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	
1	1	0	1	F	F	F	F	F	F	F	F	F	F	F	F	F	T	F	T	
1	1	1	0	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	
1	1	1	1	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	
x_0	x_1	x_2	x_3																	

With level-4 CFLOBDDs, one has $n = 8$ variables for each dimension in the full-size matrix. Thus, there are 4 A-connection variables for each dimension of the block structure, and 4 B-connection variables for each dimension of the sub-matrix for a block. Consequently, a matrix of size 256×256 is decomposed into 256 ($= 16 \times 16 = \sqrt{256} \times \sqrt{256}$) blocks, each of size $16 \times 16 = \sqrt{256} \times \sqrt{256}$.

In general, an $N \times N$ matrix is decomposed according to its $\sqrt{N} \times \sqrt{N}$ block structure, where each block is of size $\sqrt{N} \times \sqrt{N}$. With CFLOBDDs, one hopes that many of the blocks are shared among the B-connections (and possibly some blocks are even structurally similar to the block structure itself, represented by the A-connection), so that one ends up with some—hopefully small—number of subproblems γ, each of size $\sqrt{N} \times \sqrt{N}$.

The CFLOBDD decomposition discussed above is different from (i) the natural decomposition of a matrix represented via a BDD, and (ii) the decomposition used in most divide-and-conquer algorithms on matrices. Both (i) and (ii) use $\frac{n}{2} \times \frac{n}{2}$-

decompositions (and thus decompose a matrix of size 16×16 into 4 sub-matrices, each of size 8×8, and decompose a matrix of size 256×256 into 4 sub-matrices, each of size 128×128).

3.6.4.2 Vector Representation

A vector can be represented via a CFLOBDD in a manner that is similar to, but simpler than, the way matrices are represented. A vector of size $2^n \times 1$ can be represented by a CFLOBDD whose highest level is $\log n$. Suppose that V is a $2^n \times 1$ vector; a CFLOBDD representing V would have n Boolean variables $\{x_0, x_1, \ldots, x_{n-1}\}$ with the variables $\{x_0, x_1, \ldots, x_{n-1}\}$ representing the successive bits of x—the index into V.[14]

We typically use either the increasing variable ordering or decreasing variable ordering to represent vectors. (Similar to matrices, vectors of other sizes can be embedded within a larger vector of the form $2^n \times 1$.) For example, if we have a vector whose entries are defined by $\lambda x_0 x_1.(x_0 \wedge x_1)$, the vector would be as follows:

$$\begin{array}{cc|c|} \cline{3-3} 0 & 0 & F \\ 0 & 1 & F \\ 1 & 0 & F \\ 1 & 1 & T \\ \cline{3-3} x_0 & x_1 & \end{array}$$

This ordering helps in easily converting a vector of size $2^n \times 1$ to a matrix of size $2^n \times 2^n$ with interleaved-variable ordering (with the vector embedded into the matrix as the first column, and the rest of the entries set to zero).

3.6.5 Kronecker Product

When using CFLOBDDs to represent matrices on which Kronecker products will be performed, we typically use the interleaved-variable ordering. In this section,

[14]Similar to matrices, vectors of other sizes can be represented using CFLOBDDs; for instance, they can be embedded within a larger vector whose dimensions are of the form $2^n \times 1$.

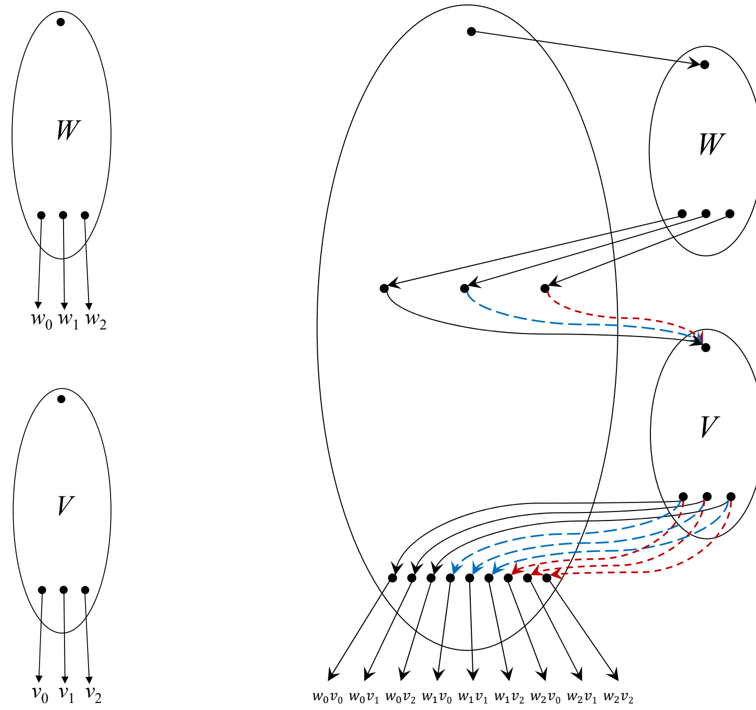


Figure 3.11: (a) and (b) Level-k CFLOBDDs for arrays W and V, respectively; (c) level-$k+1$ CFLOBDD for $W \otimes V$.

we describe two variants of Kronecker product that result in different interleavings of the index variables of the argument matrices.

3.6.5.1 Variant 1

Suppose that matrices W and V are represented by level-k CFLOBDDs with value tuples $[w_0, \ldots, w_m]$ and $[v_0, \ldots, v_n]$, respectively. To create the CFLOBDD for $W \otimes V$,

1. Create a level $k+1$ grouping that has $m+1$ middle vertices, corresponding to the values $[w_0, \ldots, w_m]$, and $(m+1)(n+1)$ exit vertices, corresponding to the terminal values

$$[w_i v_j : i \in [0..m], j \in [0..n]],$$

where the terminal values are ordered lexicographically by their (i, j) indexes;

i.e., w_0v_0, w_0v_1, ..., w_mv_{n-1}, w_mv_n. The grouping's A-connection is the proto-CFLOBDD of W, with return edges that map the ith exit vertex to middle vertex w_i.

2. For each middle vertex, which corresponds to some value w_i, $0 \leq i \leq m$, create a B-connection to the proto-CFLOBDD of V, with return edges that map the jth exit vertex to the exit vertex of the level $k+1$ grouping that corresponds to the value w_iv_j.

3. If any of the values in the sequence $[w_iv_j : i \in [0..m], j \in [0..n]]$ are duplicates, make an appropriate call on `Reduce` to fold together the classes of exit vertices that are associated with the same value, thereby creating a canonical multi-terminal CFLOBDD.

The construction through step (2) is illustrated in Fig. 3.11. Pseudo-code for the algorithm in presented in Appendix §G.

With this algorithm, if $x \bowtie y$ represents the variable ordering of W and $w \bowtie z$ represents the variable ordering of V (where $\bowtie$ denotes the operation to interleave two variable orderings), then $W \otimes V$ has the variable ordering $(x||w) \bowtie (y||z)$ (where $||$ denotes the concatenation of two sequences of variables).

3.6.5.2 Variant 2

There is a second way to perform a Kronecker product of W and V that results in a representation of $W \otimes V$ that has the variable ordering $(x \bowtie w) \bowtie (y \bowtie z)$. (As discussed in §3.8.2.4, this version of Kronecker product is useful in Simon's Algorithm.) The steps for $W \otimes V$ are as follows:

- For the CFLOBDD that represents matrix W, for every grouping of W from the top-level grouping down to level 2, create a copy of the grouping at one level greater. At level 1, create a level-2 grouping in which (i) the A-connection

Algorithm 16: ShiftToAConnectionAtLevelOne

```
    Input: Grouping g with variable ordering x ⋈ y
    Output: Grouping g' is equal to g with dummy variables w' and z' such
            that the variable ordering is ((x ⋈ w') ⋈ (y ⋈ z'))
 1  begin
 2      InternalGrouping g' = new InternalGrouping(g.level + 1);
 3      if g.level == 1 then
 4          g'.AConnection = g;
 5          g'.AReturnTuple = [1..g.numberOfExits];
 6          g'.numberOfBConnections = |g'.AReturnTuple|;
 7          for i ← 1 to g'.numberOfBConnections do
 8              g'.BConnection[i] = NoDistinctionProtoCFLOBDD(g.level);
 9              g'.BReturnTuples[i] = [i];
10          end
11          g'.numberOfExits = g.numberOfExits;
12      else
13          g'.AConnection =
             ShiftToAConnectionAtLevelOne(g.AConnection);
14          g'.AReturnTuple = g.AReturnTuple;
15          g'.numberOfBConnections = |g.AReturnTuple|;
16          for i ← 1 to g'.numberOfBConnections do
17              g'.BConnection[i] =
                 ShiftToAConnectionAtLevelOne(g.BConnection[i]);
18              g'.BReturnTuples[i] = g.BReturnTuples[i];
19          end
20          g'.numberOfExits = g.numberOfExits;
21      end
22      return RepresentativeGrouping(g');
23  end
```

is the current level-1 grouping, and (ii) the B-connections are all the level-1 no-distinction proto-CFLOBDD (Fig. 3.5(b)). In essence, this step adds dummy variables that are proxies for matrix V's variables. Alg. 16 shows the algorithm for this operation.

- For the CFLOBDD that represents matrix V, for every grouping of V from the top-level grouping down to level 2, create a copy of the grouping at one level

Algorithm 17: ShiftToBConnectionAtLevelOne

```
   Input: Grouping g with variable ordering w ⋈ z
   Output: Grouping g' is equal to g with dummy variables x' and y' such
           that the variable ordering is ((x' ⋈ w) ⋈ (y' ⋈ z))
 1 begin
 2   InternalGrouping g' = new InternalGrouping(g.level + 1);
 3   if g.level == 1 then
 4     g'.AConnection = NoDistinctionProtoCFLOBDD(g.level);
 5     g'.AReturnTuple = [1];
 6     g'.numberOfBConnections = 1;
 7     g'.BConnection[1] = ShiftToBConnectionAtLevelOne(g.level);
 8     g'.BReturnTuples[1] = g.BReturnTuples[1];
 9     g'.numberOfExits = g.numberOfExits;
10   else
11     g'.AConnection =
        ShiftToAConnectionAtLevelOne(g.AConnection);
12     g'.AReturnTuple = g.AReturnTuple;
13     g'.numberOfBConnections = |g.AReturnTuple|;
14     for i ← 1 to g'.numberOfBConnections do
15       g'.BConnection[i] =
          ShiftToAConnectionAtLevelOne(g.BConnection[i]);
16       g'.BReturnTuples[i] = [1..g.numberOfExits];
17     end
18     g'.numberOfExits = g.numberOfExits;
19   end
20   return RepresentativeGrouping(g');
21 end
```

greater. At level 1, create a level-2 grouping in which (i) the A-connection is the level-1 no-distinction proto-CFLOBDD (Fig. 3.5(b)), and (ii) the B-connection is the current level-1 grouping. In essence, this step adds dummy variables that are proxies for matrix W's variables. Alg. 17 shows the algorithm for this operation.

- Finally, combine the two newly constructed CFLOBDDs by making a call to BinaryApplyAndReduce (Alg. 11), with "Times" (multiplication) as the operation to apply to terminal values.

Algorithm 18: Kronecker Product

Input: CFLOBDDs n1, n2 with variable ordering of n1: $x \bowtie y$ and n2: $w \bowtie z$

Output: CFLOBDD n = n1 $\otimes$ n2 with variable ordering of n: $(x||w) \bowtie (y||z)$

```
1  begin
       // Add dummy variables ⟨w'_1, ··· , w'_n⟩, ⟨z'_1, ··· , z'_n⟩ such that
          variable ordering of g1 is (x ⋈ w') ⋈ (y ⋈ z')
2      CFLOBDD g1 = RepresentativeCFLOBDD(ShiftToAConnectionAtLevelOne(n1.grouping),
        n1.valueTuple);
       // Add dummy variables ⟨x'_1, ··· , x'_n⟩, ⟨y'_1, ··· , y'_n⟩ such that
          variable ordering of g2 is (x' ⋈ w) ⋈ (y' ⋈ z)
3      CFLOBDD g2 = RepresentativeCFLOBDD(ShiftToBConnectionAtLevelOne(n2.grouping),
        n2.valueTuple);
4      CFLOBDD n = BinaryApplyAndReduce(g1, g2, (op)Times);
5      return n;
6  end
```

Pseudo-code for this algorithm is given as Alg. 18.

3.6.6 Vector-to-Matrix Conversion

An important operation that is repeatedly performed in the algorithms summarized in §3.8.2 is vector-matrix multiplication. Our approach to vector-matrix multiplication is to convert a vector V of size $2^n \times 1$ into a matrix M of size $2^n \times 2^n$, where V occupies the first column, and all other entries of M are 0. We can then use the matrix-matrix multiplication algorithm discussed in §3.6.7.[15] Note that the CFLOBDD representation of V has n variables and its highest level is $\log n$, whereas the CFLOBDD for matrix M has $2n$ variables and its highest level is $\log n + 1$.

We will denote the variables in the CFLOBDD representation of V as $x = \langle x_1, x_2, \cdots, x_n \rangle$. The rows of M would use the same x variables. To represent the

[15]Matrix-vector multiplication is performed similarly.

columns of M, we introduce an extra set of n variables: $y = \langle y_1, y_2, \cdots, y_n \rangle$. As discussed in §3.6.4.1, we will use the interleaved ordering of x and y ($x \bowtie y$) to represent M; i.e., the decisions (at level 0) by A-connection groupings at level 1 are the decisions for x variables, and the decisions (at level 0) by B-connection groupings at level 1 are those for y variables.

At a high level, the algorithm for vector-to-matrix conversion involves the use of level-0 groupings (representing x variables) from V's CFLOBDD representation as the A-connection groupings at level 1 for representing M. The detailed steps of vector-to-matrix conversion are as follows:

1. Create a new CFLOBDD $c1$ of level $\log n{+}2$ from V's CFLOBDD representation c, by using level-0 groupings of c as the A-connection groupings of $c1$'s level-1 groupings and adding *Don'tCareGroupings* as B-connection groupings at level 1 similar to Alg. 17. This step creates a CFLOBDD that represents a matrix in which every column is the vector V—because all of the column variables correspond to the added *DontCareGroupings*.

2. Create another CFLOBDD $c2$ of level $\log n + 2$ that represents a matrix $Column1Matrix_n$ of size $2^n \times 2^n$ with only the first column filled with 1s and the rest with 0s.

$$Column1Matrix_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}_{2^n \times 2^n}$$

   $Column1Matrix_n$ can be recursively defined in terms of $Column1Matrix_{n/2}$ matrices of size $2^{n/2} \times 2^{n/2}$. This property allows $c2$ to both be created and

Algorithm 19: Vector-to-Matrix Conversion

Input: CFLOBDD n representing vector V
Output: CFLOBDD n' representing matrix M obtained by padding V with zeros.

```
1 begin
2     CFLOBDD g1 = RepresentativeCFLOBDD(ShiftToBConnectionAtLevelOne(n.grouping),
        n.valueTuple);
3     CFLOBDD g2 = Column1Matrix(n.grouping.level + 1);
4     CFLOBDD n = BinaryApplyAndReduce(g1, g2, (op)Times);
5     return n;
6 end
```

represented efficiently.

$$Column1Matrix_n = \begin{cases} \begin{bmatrix} Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \vdots & \vdots & \ddots & \vdots \\ Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \end{bmatrix}_{2^n \times 2^n} & \\ \quad = Column1Matrix_{n/2} \otimes Column1Matrix_{n/2} & n > 1 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & n = 1 \end{cases}$$

Here, O_j represents the all-zero matrix of size $2^j \times 2^j$. An algorithm for the efficient construction of $Column1Matrix_n$ can be found in Appendix §H.

3. Finally multiply $c1$ and $c2$ pointwise by calling BinaryApplyAndReduce.

Pseudo-code for this algorithm is given as Alg. 19.

Example 3.4. We illustrate the steps of the algorithm using the following example: Consider the vector $V = \begin{bmatrix} 2 \\ 3 \\ 5 \\ 0 \end{bmatrix}$ and matrix $M = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$. The goal is to convert V to M. Steps 1 and 2 construct intermediate matrices M_1 and M_2, defined as follows: $M_1 = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ and $M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$. Finally, the result of converting vector V to

a matrix is $M = M_1 * M_2 = \begin{bmatrix} 2&2&2&2 \\ 3&3&3&3 \\ 5&5&5&5 \\ 0&0&0&0 \end{bmatrix} * \begin{bmatrix} 1&0&0&0 \\ 1&0&0&0 \\ 1&0&0&0 \\ 1&0&0&0 \end{bmatrix} = \begin{bmatrix} 2&0&0&0 \\ 3&0&0&0 \\ 5&0&0&0 \\ 0&0&0&0 \end{bmatrix}$, where "*" denotes pointwise matrix multiplication.

3.6.7 Matrix Multiplication

Matrix multiplication is one of the most important operations on matrices. This section discusses how to perform matrix multiplication when the matrices are represented as CFLOBDDs using the interleaved-variable ordering.

The multiplication algorithm for CFLOBDDs presented here is similar to the standard $O(N^3)$ algorithm for multiplying two $N \times N$ matrices. There is a potential for savings because each of the argument CFLOBDDs may have a large number of shared substructures, and function caching can be used to detect when a sub-problem has already been performed, in which case the proto-CFLOBDD for the answer can be returned immediately.

Our starting point is the observation that when the interleaved-variable ordering is used, at top level the A-connection of a CFLOBDD-represented matrix M captures commonalities in the $\sqrt{N} \times \sqrt{N}$ block structure of M, and the B-connections represent sub-matrices of M of size $\sqrt{N} \times \sqrt{N}$. By analogy with other kinds of multi-terminal CFLOBDDs, at top-level one can think of the A-connection as a multi-terminal CFLOBDD whose value tuple is the sequence of B-connections—roughly, the A-connection is a $\sqrt{N} \times \sqrt{N}$ matrix with $\sqrt{N} \times \sqrt{N}$-matrix-valued leaves.

Suppose that P and Q are two $N \times N$ matrices represented by CFLOBDDs C_P and C_Q, respectively. The respective top-level A-connections, A_P and A_Q, are matrices of size $\sqrt{N} \times \sqrt{N}$ with matrix-valued cells of size $\sqrt{N} \times \sqrt{N}$. To multiply P and Q, we first recursively multiply A_P and A_Q. This operation defines which cells of A_P and A_Q get multiplied and added—and the answer is returned as a collection of symbolic expressions (of a form that will be described shortly). Using this information, we recursively call matrix multiplication and matrix addition on the B-connections, as appropriate. For the base case of the recursion—namely, level 1,

which represents matrices of size 2×2—we can enumerate all the individual cases of possible matrix structures (i.e., the patterns of which cells hold equal values), and build the CFLOBDDs that result from a matrix multiplication in each case.

We now describe how the symbolic information mentioned above is organized, and how operations of addition and multiplication are performed on the data type in which the symbolic information is represented. The challenge that we face is that at all levels below top-level, we do not have access to a *value* for any cell in a matrix. However, we can use the exit vertices as variables.

Example 3.5. Suppose that we are multiplying two level-1 groupings that, when considered as 2×2 matrices over their respective exit vertices $[ev_1, ev_2]$ and $[ev'_1, ev'_2, ev'_3]$, have the forms shown on the left

$$\begin{bmatrix} ev_1 & ev_1 \\ ev_2 & ev_2 \end{bmatrix} \times \begin{bmatrix} ev'_1 & ev'_2 \\ ev'_1 & ev'_3 \end{bmatrix} = \begin{bmatrix} ev_1 ev'_1 + ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ ev_2 ev'_1 + ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix} = \begin{bmatrix} 2 ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ 2 ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix} \tag{3.6}$$

Each entry in the right-hand-side matrix can be represented by a set of triples, e.g.,

$$\begin{bmatrix} \{[(1,1),2]\} & \{[(1,2),1],[(1,3),1]\} \\ \{[(2,1),2]\} & \{[(2,2),1],[(2,3),1]\} \end{bmatrix}$$

and when listed in exit-vertex order for the interleaved-variable order, we have

$$[\{[(1,1),2]\}, \{[(1,2),1],[(1,3),1]\}, \{[(2,1),2]\}, \{[(2,2),1],[(2,3),1]\}]. \tag{3.7}$$

Now suppose that the two matrices are sub-matrices of level-2 groupings connected by ReturnTuples $rt = [5,2]$ and $rt' = [6,1,2]$, respectively. Then applying $\langle rt, rt' \rangle$ to Eqn. (3.7) results in

$$[\{[(5,6),2]\}, \{[(5,1),1],[(5,2),1]\}, \{[(2,6),2]\}, \{[(2,1),1],[(2,2),1]\}]. \tag{3.8}$$

We call the objects shown in Eqns. (3.7) and (3.8) *MatMultTuples*. By this device, the answer to a matrix-multiplication sub-problem (whether from A-connections or B-connections, and at any level ≥ 1) can be treated as a multi-terminal CFLOBDD whose value tuple is a MatMultTuple.

Semantics of MatMultTuples. An alternative view of MatMultTuples comes from the right-hand matrix in Eqn. (3.6): a MatMultTuple is a sequence of bilinear polynomials over the exit vertices of two groupings. We will represent a bilinear polynomial p as a map from exit-vertex pairs to the corresponding coefficient. (The pairs for which the coefficient is nonzero are called the *support* of p. In examples, we show only map entries that are in the support.) In particular, suppose that g_1 and g_2 are two groupings at the same level, with exit-vertex sets EV and EV'. Each entry of a MatMultTuple is of type $BP_{EV,EV'} \stackrel{\text{def}}{=} (EV \times EV') \to \mathbb{N}$. (We will drop the subscripts on BP if the exit-vertex sets are understood.)

To perform linear arithmetic on bilinear polynomials, we define

$$\begin{array}{rr}
0_{BP} : BP & 0_{BP} \stackrel{\text{def}}{=} \lambda(ev, ev').0 \\
+ : BP \times BP \to BP & bp_1 + bp_2 \stackrel{\text{def}}{=} \lambda(ev, ev').bp_1(ev, ev') + bp_2(ev, ev') \\
* : \mathbb{N} \times BP \to BP & n * bp \stackrel{\text{def}}{=} \lambda(ev, ev').n * bp(ev, ev')
\end{array}$$

By considering a ReturnTuple to be a map from one exit-vertex set to another, this notation allows us to give a second account of the transformation from Eqn. (3.7) to Eqn. (3.8). For instance, let $rt = [1 \mapsto 5, 2 \mapsto 2]$ and $rt' = [1 \mapsto 6, 2 \mapsto 1, 3 \mapsto 2]$. Consider the second element of Eqn. (3.7): $bp = \{[(1,2),1],[(1,3),1]\} = [(1,2) \mapsto 1, (1,3) \mapsto 1]$. Then the transformation of bp induced by rt and rt' can be expressed as follows (where Eqn. (3.9) expresses the general case):

$$\begin{aligned}
\langle rt, rt' \rangle(bp) &\stackrel{\text{def}}{=} \{(rt(ev), rt'(ev')) \mapsto bp(ev, ev') \mid ev \in EV, ev' \in EV'\} \qquad (3.9) \\
&= \{(rt(1), rt'(2)) \mapsto bp(1,2), (rt(1), rt'(3)) \mapsto bp(1,3)\} \\
&= \{(5,1) \mapsto 1, (5,2) \mapsto 1\}
\end{aligned}$$

At top level, we need a similar operation for the value induced by a pair of value tuples $\langle vt, vt' \rangle$ (where a value tuple is treated as a map of type $EV \to \mathbb{V}$ for a value space $\mathbb{V}$ that supports + and *):

$$\langle vt, vt' \rangle(bp) \stackrel{\text{def}}{=} \sum \{bp(ev, ev') * vt(ev) * vt'(ev') \mid ev \in EV, ev' \in EV'\} \qquad (3.10)$$

Algorithm 20: Matrix Multiplication

```
   Input: CFLOBDDs n1, n2
   Output: CFLOBDD n = n1 × n2
 1 begin
 2   Grouping×MatMultTuple [g,m] =
       MatrixMultOnGrouping(n1.grouping, n2.grouping);
 3   ValueTuple v_tuple = [];
 4   for i ← 1 to |m| do
 5     Value v = ⟨n1.valueTuple, n2.valueTuple⟩(m(i));
 6     v_tuple = v_tuple || v;
 7   end
 8   Tuple×Tuple [inducedValueTuple, inducedReductionTuple] =
       CollapseClassesLeftmost(v_tuple);
 9   g = Reduce(g, inducedReductionTuple);
10   CFLOBDD n = RepresentativeCFLOBDD(g, inducedValueTuple);
11   return n;
12 end
```

Algs. 20, 21, and 22 give pseudo-code for the matrix-multiplication algorithm. Algs. 21 and 22 operates on two Groupings g1 and g2 at some level l, returning a (Grouping, MatMultTuple) pair (g, m). Algs. 21 and 22 works symbolically, first considering the symbolic product of g1.AConnection and g2.AConnection (line [5]). The MatMultTuple ma returned from this call is really a list of directives: each directive is a bilinear polynomial used to create one of the BConnections of g. The workhorse computation—lines [9]–[26]—sets g.BConnections[i] to the (symbolic) weighted dot product

$$\sum_{((k_1,k_2),v)\in \text{ma}(i)} v * \text{g1.BConnections}[k_1] * \text{g2.BConnections}[k_2].$$

To perform this computation, an auxiliary multi-terminal CFLOBDD curr_cflobdd is used to accumulate the (symbolic) weighted dot product. Note that the valueTuple of curr_cflobdd is a MatMultTuple; thus, curr_cflobdd is essentially a matrix, each element of which is a bilinear polynomial—i.e., a directive saying how to compute the value of that element from a set of pairs of (as yet unknown) values.

```
Algorithm 21: MatrixMultOnGrouping
   Input: Groupings g1, g2
   Output: Grouping×MatMultTuple [g,m] such that g = g1 × g2
1  begin
2  |  if g1.level == 1 then    // Base Case: matrices of size 2 × 2
   |  |  // Construct a level 1 Grouping that reflects which
   |  |     cells of the product hold equal entries in the
   |  |     output MatMultTuple
3  |  end
4  |  InternalGrouping g = new InternalGrouping(g1.level);
5  |  Grouping×MatMultTuple [aa,ma] =
   |   MatixMultOnGrouping(g1.AConnection, g2.AConnection);
6  |  g.AConnection = aa; g.AReturnTuple = [1..|ma|];
   |   g.numberOfBConnections = |ma|;
   |  // Interpret ma to (symbolically) multiply and add
   |     BConnections
7  |  MatMultTuple m = [];
8  |  for i ← 1 to |ma| do       // Interpret i^th BP in ma to create
   |   g.BConnections[i]
   |  |  // Set g.BConnections[i] to the (symbolic) weighted dot
   |  |     product
   |  |     Σ_{((k1,k2),v)∈ma(i)} v * g1.BConnections[k1] * g2.BConnections[k2]
9  |  |  CFLOBDD curr_cflobdd = ConstantCFLOBDD(g1.level, [0_BP]);
10 |  |  for ((k1, k2), v) ∈ ma(i) do
11 |  |  |  Grouping×MatMultTuple [bb,mb] =
   |  |  |   MatrixMultOnGrouping(g1.BConnections[k1],
   |  |  |   g2.BConnections[k2]);
12 |  |  |  MatMultTuple mc = [];
13 |  |  |  for j ← 1 to |mb| do
14 |  |  |  |  BP bp =
   |  |  |  |   ⟨g1.BReturnTuples[k1], g2.BReturnTuples[k2]⟩(mb(j));
15 |  |  |  |  mc = mc || bp;
   |  |  // Continued in Alg. 22
```

Alg. 20 multiplies two matrices, n1 and n2, represented as CFLOBDDs. It initiates the process at top level by calling MatrixMultGrouping on n1.grouping and n2.grouping (line [3]). It then uses the returned MatMultTuple as a list of directives, similar to the way MatMultTuples are used in MatrixMultGrouping. The difference is

Algorithm 22: MatrixMultOnGrouping (cont.)

```
16
17
18
19          Tuple×Tuple [inducedMatMultTuple, inducedReductionTuple]
             = CollapseClassesLeftmost(mc);
20          bb = Reduce(bb, inducedReductionTuple);
21          CFLOBDD n = RepresentativeCFLOBDD(bb,
             inducedMatMultTuple);
22          curr_cflobdd = curr_cflobdd + v ∗ n ;        // Accumulate
             symbolic sum
23        end
24        g.BConnection[i] = curr_cflobdd.grouping;
25        g.BReturnTuples[i] = curr_cflobdd.valueTuple;
26        m = m || curr_cflobdd.valueTuple;
27      end
28      g.numberOfExits = |m|;
29      Tuple×Tuple [inducedMatMultTuple, inducedReductionTuple] =
         CollapseClassesLeftmost(m) ;
30      g = Reduce(g, inducedReductionTuple);
31      return [RepresentativeGrouping(g), m];
32  end
```

that at top level one has access to the actual values of the argument matrices, namely, n1.valueTuple and n2.valueTuple. Thus, in Alg. 20 the elements of MatMultTuple m are interpreted as directives to perform a *concrete* weighted dot product (lines [4]–[7]), using Eqn. (3.10) in line [5], thereby computing the values of the elements of the answer matrix .

3.6.8 Path Counting and Sampling

A CFLOBDD whose terminal values are non-negative numbers can be used to represent a discrete distribution over the set of assignments to the Boolean variables. An assignment—or equivalently, the corresponding matched path in the CFLOBDD—is considered to be an elementary event. The "weight" of the elementary event is the terminal value. The probability of a matched path p is the weight of p divided by the total weight of the CFLOBDD—the sum of the weights obtained by following each of the CFLOBDD's matched paths. Fortunately, it is possible to compute the aforementioned denominator by computing, for each of the terminal values, the number of matched paths that lead to that terminal value (§3.6.8.1). With those numbers in hand, it is then possible to sample an assignment/path according to the distribution that the CFLOBDD represents (§3.6.8.2).

The same approach can be used for CFLOBDDs whose terminal values are complex numbers, except that the weight of a matched path is the modulus of the terminal value. This approach is used in the application of CFLOBDDs to quantum simulation (§3.8 and §3.9.2.2).

3.6.8.1 Path Counting

Recall that every terminal value is connected to one exit vertex of the top-level grouping of the CFLOBDD. Every exit vertex of a grouping is, in turn, connected to exit vertices of internal groupings. Therefore, to compute the number of matched paths for every terminal value, we need to compute the path-counts from the entry

vertex of a grouping to every exit vertex of that grouping, for every grouping in the CFLOBDD. For each grouping g, we would like to compute a vector of path-counts, in which the ith element is the number of matched paths from g's entry vertex to the ith exit vertex of g. To compute this information, we can break it down into (i) computing the number of matched paths from g's entry vertex to g's middle vertices; (ii) computing the number of matched paths from g's middle vertices to g's exit vertices; and (iii) combining this information to obtain the number of matched paths from g's entry vertex to g's exit vertices.

Consider a Grouping g at level l with e exit vertices. Suppose that *g.AConnection* has p exit vertices, *g.BConnections*[j] has k_j exit vertices, and let *g.BReturnTuples*[j] be the return edges from *g.BConnections*[j]'s exit vertices to g's exit vertices. For step (i), we recursively compute the path-counts for *g.AConnection*, which yields a vector of path-counts v_A of size $1 \times p$. Step (ii) creates a matrix M_B of size $p \times e$, in which the jth row is the vector of path-counts from the jth middle vertex of g to g's exit vertices. Step (iii) is the vector-matrix multiplication $v_A \times M_B$, which yields g's path-count vector, of size $1 \times e$. The base-case path-count vectors are $[1, 1]$ for a *ForkGrouping* and [2] for a *DontCareGrouping*.

Because the exit vertices of *g.BConnections*[j] are connected to g's exit vertices via *g.BReturnTuples*[j], the jth row of M_B is the product of the path-count vector for *g.BConnections*[j] (of size $1 \times k_j$) and a "permutation matrix" $PM^{g.BReturnTuples[j]}$ (of size $k_j \times e$). Each entry of PM is either 0 or 1; each row must have exactly one 1; and each column must have at most one 1.

Alg. 23 shows pseudo-code for the path-counting algorithm. The path-counts are stored in a Tuple *numPathsToExit* in the Grouping data structure. Lines [13]–[20] perform the vector-matrix multiplication discussed above. In practice, because the number of path-counts increases double exponentially in the number of levels, we only store the log-values of the path-counts. It is also important for the implementation of the algorithm to perform function caching (§3.4.3) so that each grouping at each level

Algorithm 23: CountPaths

```
   Input: Grouping g
 1 begin
 2   if g.level == 0 then
 3     if g == DontCareGrouping then
 4       g.numPathsToExit = [2];
 5     else // g == ForkGrouping
 6       g.numPathsToExit = [1,1];
 7     end
 8   else
 9     CountPaths(g.AConnection);
10     for i ← 1 to g.numberOfBConnections do
11       CountPaths(g.BConnection[i]);
12     end
13     g.numPathsToExit = [1..|g.numberOfExits|];
14     for i ← 1 to g.numberOfBConnections do
15       for j ← 1 to g.BConnection[i].numberOfExits do
16         k = BReturnTuples[i](j);
17         g.numPathsToExit[k] +=
18           g.AConnection.numPathsToExit[i] *
             g.BConnection[i].numPathsToExit[j];
19       end
20     end
21   end
22 end
```

is visited only once. When function caching is employed, Alg. 23 visits each grouping, and hence each vertex and edge of the CFLOBDD, exactly once; consequently, the cost of the path-counting operation is bounded by the size of the argument CFLOBDD.

This definition can also be stated equationally, in a form similar to the denotational semantics given in §3.5, where the expression in large brackets represents M_B.

$$numPathsToExit^{g}_{1\times e} = \begin{cases} [1,1]_{1\times 2} & \text{if g} = \mathit{ForkGrouping} \\ [2]_{1\times 1} & \text{if g} = \mathit{DontCareGrouping} \\ numPathsToExit^{g.AConnection}_{1\times p} \times \left[\begin{array}{c} \vdots \\ numPathsToExit^{g.BConnections[j]}_{1\times k_j} \\ \times PM^{g.BReturnTuples[j]}_{k_j \times e} \\ \vdots \end{array} \right]_{\substack{p\times e \\ j\in\{1..p\}}} & \text{otherwise} \end{cases}$$

Example 3.6. For the five proto-CFLOBDDs depicted in Fig. 3.7, the vectors of path-counts are computed as follows (read top-to-bottom by level):

level 2	level 1	level 0
$\begin{bmatrix}9 & 7\end{bmatrix} = \begin{bmatrix}3 & 1\end{bmatrix} \times \begin{bmatrix}3 & 1 \\ 0 & 4\end{bmatrix}$	$\begin{bmatrix}3 & 1\end{bmatrix} = \begin{bmatrix}1 & 1\end{bmatrix} \times \begin{bmatrix}2 & 0 \\ 1 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 1\end{bmatrix}$
	$[4] = [2] \times [2]$	$[2]$

3.6.8.2 Sampling an Assignment

Our goal is to sample a matched path from the distribution of matched paths of a given CFLOBDD, and return the corresponding assignment. As in §3.2.3.4, we assume that an assignment is an array of Booleans, whose entries—starting at index-position 1—are the values of successive variables. The concatenation of two such arrays a_1 and a_2 is denoted by $a_1||a_2$.

To explain how to sample from the set of assignments, we argue in terms of the structure of the corresponding matched paths. If the distribution of matched

paths was given as a vector of weights, $W = [w_1..w_{2^{2^l}}]$, as one would have in the corresponding decision tree, the probability of selecting the pth matched path is given by

$$Prob(p) = \frac{w_p}{\sum_{i=1}^{2^{2^l}} w_i}. \tag{3.11}$$

In a CFLOBDD that represents a distribution, we do not have access to W directly. Suppose that $W' = [w'_1, \ldots, w'_K]$ is the vector of terminal values of the CFLOBDD. The K values of W' are exactly the K different values that appear in W; however, many matched paths that start at the top-level entry vertex lead to the same terminal value, say w'_m. Fortunately, the path-counting method from §3.6.8.1 provides us with part of what is needed via *NumPathsToExit* of the top-level grouping.

$$\sum_{i=1}^{2^{2^l}} w_i = \sum_{j=1}^{K} w'_j \times numPathsToExit[j]$$

Thus, while Eqn. (3.11) becomes

$$Prob(p) = \frac{w_p}{\sum_{j=1}^{K} w'_j \times numPathsToExit[j]}$$

this observation gives us no guidance about how to select a matched path p with that probability.

Rather than selecting a single matched path immediately, what we can do instead is to select the entire set of matched paths that reach a given terminal value. This selection can be done by sampling from the exit vertices of the top-level grouping according to the probability distribution

$$Prob'(\text{Path ends at terminal value } w'_t) = \frac{w'_t \times numPathsToExit[t]}{\sum_{j=1}^{K} w'_j \times numPathsToExit[j]} \tag{3.12}$$

The result of this sampling step is the index of an exit vertex of the top-level grouping, which will be used for further sampling among the (indirectly) "retrieved" set of matched paths. What remains to be done is to uniformly sample a matched path from that set, and return the assignment that corresponds to that matched path.

To achieve this goal, we take advantage of the structure of matched paths to break the assignment/path-sampling problem down to a sequence of smaller assignment/path-sampling problems that can be performed recursively. At each grouping g visited by the algorithm, the goal is to uniformly sample a matched path from the set of matched paths $P_{g,i}$ (in the proto-CFLOBDD headed by g) that lead from g's entry vertex to a specific exit vertex i of g.

Consider a grouping g and a given exit vertex i. For each middle vertex m of g, there is some number of matched paths—possibly 0—from the entry vertex of g that pass through m and eventually reach exit vertex i. Those numbers of matched paths, when divided by $|P_{g,i}|$, represent a distribution D_i on the set of g's middle vertices. Consequently, the first step toward uniformly sampling a matched path from the set $P_{g,i}$ is to sample the index of a middle vertex of g according to distribution D_i. Call the result of that sampling step m_{index}. Thus, to sample a matched path from the entry vertex of g to exit vertex i, we (i) sample a middle vertex of g according to D_i to obtain m_{index}; (ii) uniformly sample a matched path from $g.AConnection$ with respect to the exit vertex of $g.AConnection$ that returns to m_{index}; (iii) uniformly sample a matched path from $g.BConnections[m_{index}]$ with respect to whichever of its exit vertices is connected to the ith exit vertex of g; and (iv) concatenate the assignments obtained from steps (ii) and (iii).

Only the B-connections of g whose exit vertices are connected to i (the distinguished exit vertex of g) can contribute to the paths leading to i, and hence we need to select a middle vertex from among those for which the B-connection grouping can lead to i. For such an i-connected B-connection grouping k, let $(g.BReturnTuples[k])^{-1}[i]$ denote the exit vertex of $g.BConnections[k]$ that leads to i; i.e., $\langle j, i \rangle \in g.BReturnTuples[k] \Leftrightarrow (g.BReturnTuples[k])^{-1}[i] = j$.

The path-counts for the number of matched paths of g's B-connections (available via the vector *NumPathsToExit* for each of g's B-connections, denoted by, e.g., $numPathsToExit^{g.BConnections[k]}$) only considers matched paths from g's middle ver-

tices to g's exit vertices. However, to sample m_{index} correctly, we need to consider *all* of the matched paths from g's entry vertex to g's exit vertex i. Hence, we multiply the number of matched paths from g's entry vertex to a middle vertex of g (of interest to us because it is connected to a B-connection that is connected to i), denoted by, e.g., $numPathsToExit^{g.AConnection}[k]$, to the number of matched paths from that same middle vertex to g's exit vertex i. Thus, the probability associated with a given m_{index} is as follows (where $g.A$ denotes $g.AConnection$, $g.B[k]$ denotes $g.BConnections[k]$, and $g.BRT$ denotes $g.BReturnTuples$):

$$Prob(m_{index}) = \frac{\begin{array}{l} numPathsToExit^{g.A}[m_{index}] \times \\ \qquad numPathsToExit^{g.B[m_{index}]}[(g.BRT[m_{index}])^{-1}[i]] \end{array}}{g.numPathsToExit[i]} \tag{3.13}$$

Example 3.7. Consider the CFLOBDD depicted in Fig. 3.7, and suppose that the goal is to sample a matched path that leads to terminal value T. From Ex. 3.6, we know that (i) the outermost grouping has 7 matched paths that lead to T, and (ii) *NumPathsToExit* is $[3, 1]$ and $[4]$ for the upper and lower level-1 groupings, respectively. Both of the outermost grouping's middle vertices have return edges that lead to T; thus, from Eqn. (3.13), we should sample the middle vertices with probabilities

$$\begin{aligned} Prob(m_{index} = 1) &= \tfrac{[3,1][1] \times [3,1][2]}{7} = \tfrac{3 \times 1}{7} = \tfrac{3}{7} \\ Prob(m_{index} = 2) &= \tfrac{[3,1][2] \times [4][1]}{7} = \tfrac{1 \times 4}{7} = \tfrac{4}{7} \end{aligned}$$

Once m_{index} has been selected in accordance with Eqn. (3.13), we recursively sample a matched path—and its assignment a_A—from $g.AConnection$ with respect to exit vertex m_{index} (step (ii)). We also recursively sample a matched path—and its assignment a_B—from $g.BConnection[m_{index}]$ with respect to the exit vertex $(g.BReturnTuples[m_{index}])^{-1}[i]$ that leads to g's exit vertex i (step (iii)). Step (iv) produces the assignment $a = a_A || a_B$.

As for the base cases of the recursion, for a *DontCareGrouping*, we randomly choose one of the paths 0 or 1 with probability 0.5, returning the assignment "0"

Algorithm 24: Sample an Assignment from a CFLOBDD

```
1 Algorithm SampleAssignment(n)
    Input: CFLOBDD n
    Output: Assignment sampled from n according to n.valueTuple
2   begin
3       i ← Sample(n.valueTuple); // Sample terminal-value index i
         via Eqn. (3.12)
4       Assignment a = SampleOnGroupings(n.grouping, i);
5       return a;
6   end
7 end
```

or "1" accordingly; for a *ForkGrouping*, the designated exit vertex—either 1 or 2—specifies a unique assignment: "0" or "1," respectively.

Algs. 24 and 25 give pseudo-code for the algorithm for sampling an assignment from a CFLOBDD.

For a CFLOBDD at level l, the sampling operation involves constructing an assignment of size 2^l. Hence, the cost of sampling is at least as large as the size of the sampled assignment. However, the size of the argument CFLOBDD also influences the cost of sampling; although not every grouping of the CFLOBDD is necessarily visited when sampling an assignment, we can say that the cost of the sampling operation is bounded by $\mathcal{O}(\max(2^l, \text{size of argument CFLOBDD}))$.

3.7 Relations efficiently represented by CFLOBDDs

In this section, we prove that there exists an exponential separation between CFLOBDDs and BDDs. We establish this result using three relations that can be efficiently represented by CFLOBDDs. Note that we do not assume any specific variable ordering when discussing the sizes of BDDs for the functions used to prove the separation. We use node counts in BDDs, and vertex counts and edge counts in CFLOBDDs as a proxy for memory. (Recall from footnote 3 that we use the term

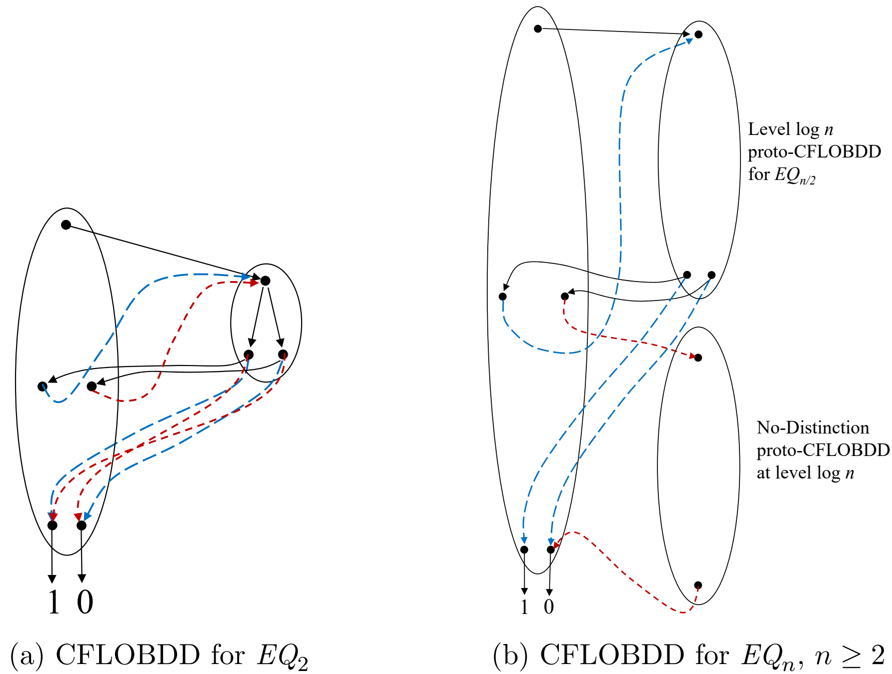
Algorithm 25: Sample an Assignment from a CFLOBDD (cont.)

```
1  SubRoutine SampleOnGroupings(g, i)
     Input: Grouping g, Exit index i
     Output: Assignment sampled from g, corresponding to one of the
             paths leading to exit i
2    begin
3      if g.level == 0 then
4        if g == DontCareGrouping then
5          return (random() % 2) ? "1" : "0";
6        else // g == ForkGrouping so i ∈ [1,2]
7          return (i == 1) ? "0" : "1"
8        end
9      end
10     Tuple PathsLeadingToI = [];
11     for j ← 1 to g.numberOfBConnections do // Build path-count
         tuple from which to sample
12       if i ∈ g.BReturnTuples[j] then        // if j^th B-connection
           leads to i
13         PathsLeadingToI = PathsLeadingToI ||
             (g.AConnection.numPathsToExit[j] *
             g.BConnections[j].numPathsToExit[k]), where i =
             BReturnTuples[j](k);
14       end
15     end
16     m_index ← Sample(PathsLeadingToI) ;  // Sample middle-vertex
         index m_index
17     Assignment a = SampleOnGroupings(g.AConnection, m_index) ||
         SampleOnGroupings(g.BConnection[m_index], k), where i =
         BReturnTuples[m_index](k);
18     return a;
19   end
20 end
```

"node" solely for BDDs, whereas "groupings" and "vertices" (depicted as the dots inside groupings) refer to CFLOBDDs.)

Remark. Recently, Zhi and Reps (2025) obtained a characterization of relative sizes in the opposite direction (i.e., a bound on CFLOBDD size as a function of BDD size, for all BDDs). They showed that for every BDD B of size $|B|$, there is a corresponding CFLOBDD C, which uses the same variable ordering, of size $O(|B|^3)$. They also showed that the bound is tight: there is a family of functions for which $|C|$ grows as $\Omega(|B|^3)$.

3.7.1 The Equality Relation EQ_n



(a) CFLOBDD for EQ_2

(b) CFLOBDD for EQ_n, $n \geq 2$

Figure 3.12

Definition 3.4. The equality relation $EQ_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \rightarrow \{0,1\}$ on variables $(x_0 \cdots x_{n/2-1})$ and $(y_0 \cdots y_{n/2-1})$ is the relation $EQ_n(X,Y) \overset{\text{def}}{=} \Pi_{i=0}^{n/2-1}(x_i \Leftrightarrow y_i) = \Pi_{i=0}^{n/2-1}(\bar{x}_i\bar{y}_i \vee x_i y_i)$.

Theorem 3.3 Exponential separation for the equality relation**.** For $n = 2^l$, where $l \geq 1$, EQ_n can be represented by a CFLOBDD with $\mathcal{O}(\log n)$ vertices and edges. In contrast, a BDD that represents EQ_n requires $\Omega(n)$ nodes.

Proof.

CFLOBDD Claim. We claim that with the interleaved-variable ordering $\langle x_0, y_0, \ldots, x_{n/2-1}, y_{n/2-1} \rangle$, the CFLOBDD representation of EQ_n uses $\mathcal{O}(\log n)$ groupings, each of constant size (and hence uses $\mathcal{O}(\log n)$ vertices and edges in total). Diagrams that illustrate the CFLOBDD representation are shown in Fig. 3.12. For EQ_2 (i.e., $l = 1$), the representation is shown in Fig. 3.12a. This CFLOBDD has two groupings, a fork grouping at level 0 and a level-1 grouping. In total, it has eight vertices and eleven edges. Note that the "success" and "failure" exits at level 1 are the left and right exits, respectively.

For EQ_n with $n > 2$ (i.e., $l > 1$), the representation is defined inductively, following the pattern shown in Fig. 3.12b. For all i, $1 \leq i \leq l = \log n$, a level-(i+1) grouping of the form shown at the outermost level of Fig. 3.12b has an A-connection and—from the leftmost middle vertex—a B-connection to the proto-CFLOBDD for $EQ_{2^{i-1}}$. The leftmost exit vertex of the proto-CFLOBDD is the "success exit" for testing equality on 2^{i-1} pairs of variables.

- When called via the level-(i+1) grouping's A-connection, the proto-CFLOBDD tests equality on the variable pairs $\{(x_0, y_0), \ldots, (x_{2^{i-1}-1}, y_{2^{i-1}-1})\}$.
- When called from the level-(i+1) grouping's leftmost middle vertex, the proto-CFLOBDD tests equality on the variable pairs $\{(x_{2^{i-1}}, y_{2^{i-1}}), \ldots, (x_{2^i-1}, y_{2^i-1})\}$.

The right exit vertex of the proto-CFLOBDD is the "failure exit." When the proto-CFLOBDD has been called from the level-(i+1) grouping's A-connection, the return edge is the matching solid edge to the rightmost middle vertex of the level-(i+1) grouping. This vertex signifies that there has been an equality mismatch on some variable pair in $\{(x_0, y_0), \ldots, (x_{2^{i-1}-1}, y_{2^{i-1}-1})\}$, and so its B-connection is to the no-distinction proto-CFLOBDD of level i, whose one exit vertex returns to the rightmost ("failure") exit vertex of the level-(i+1) grouping.

The level-(i+1) grouping has five vertices and eight edges, and the level-i grouping of the no-distinction proto-CFLOBDD at level i has three vertices and four edges (see Fig. 3.5d). Consequently, each of the $\log n + 1$ levels of the CFLOBDD for EQ_n contributes a constant number of vertices and edges, independent of i, and thus the total number of vertices and edges is $\mathcal{O}(\log n)$.

BDD Claim. Now consider a BDD representation for EQ_n. We claim that regardless of the variable order used, a BDD requires at least n nodes, one node for each argument variable.

We prove this claim by contradiction. Suppose that there is some BDD B for EQ_n that does not need at least one node for each variable. Let $\mathcal{T}$ denote the "all-true" assignment of variables; i.e., $\mathcal{T} \stackrel{\text{def}}{=} \forall k \in \{0..n/2-1\}, x_k \mapsto T, y_k \mapsto T$. There are three possible situations:

- Case 1: B does not have variable y_k, for some $k \in \{0..n/2-1\}$. Now, consider two assignments of variables: $A_1 \stackrel{\text{def}}{=} \mathcal{T}$ and $A_2 \stackrel{\text{def}}{=} \mathcal{T}[y_k \mapsto F]$ (i.e., A_2 is A_1 with y_k updated to F). Because B does not depend on y_k, the function represented by B maps both A_1 and A_2 to the same value (either 0 or 1), which violates the definition of the equality relation EQ_n (i.e., $EQ_n[A_1] = 1$ and $EQ_n[A_2] = 0$). Consequently, none of the y variables can be dropped individually.

- Case 2: Using an argument completely analogous to Case 1, we can show that none of the x variables can be dropped individually.

- Case 3: B depends on neither x_k nor y_k. Consider the following four assignments: $A_1 = \mathcal{T}$, $A_2 = \mathcal{T}[y_k \mapsto F]$, $A_3 = \mathcal{T}[x_k \mapsto F]$, and $A_4 = \mathcal{T}[x_k \mapsto F][y_k \mapsto F]$. Because B does not depend on either x_k or y_k, the function represented by B maps all four assignments to the same value (either 0 or 1), which violates the definition of the equality relation EQ_n (i.e., $EQ_n[A_1] = EQ_n[A_4] = 1$ and $EQ_n[A_2] = EQ_n[A_3] = 0$).

Because (i) no y_k can be dropped individually, (ii) no x_k can be dropped individually, and (iii) no (x_k, y_k) pair can be dropped together, B—and hence any BDD representation for EQ_n—requires $\Omega(n)$ nodes. □

3.7.2 The Hadamard Relation H_n

The Hadamard Relation represents the family of Hadamard Matrices discussed in §2.1.1 and §3.2.4. The Hadamard matrices play a role in many quantum algorithms, including the seven that are used in §3.9.2.2 to evaluate the effectiveness of CFLOBDDs for simulating quantum circuits (namely, GHZ, BV, DJ, Simon's algorithm, QFT, Shor's algorithm, and Grover's algorithm). See §3.8.2 and §3.9.2.2.

Theorem 3.4 Exponential separation for the Hadamard relation**.** The Hadamard Relation $H_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \rightarrow \{1,-1\}$ between variable sets $(x_0 \cdots x_{n/2})$ and $(y_0 \cdots y_{n/2})$, where $n = 2^l$, can be represented by a CFLOBDD with $\mathcal{O}(\log n)$ vertices and edges. In contrast, a BDD that represents H_n requires $\Omega(n)$ nodes.

Proof.

CFLOBDD Claim. As shown in §3.2.4, each matrix $H_n \in \mathcal{H}$, where $n = 2^l$ can be represented by a CFLOBDD with $\mathcal{O}(l)$ vertices and edges—i.e., with $\mathcal{O}(\log n)$ space.

BDD Claim. We claim that regardless of the variable ordering, the BDD representation for H_n requires at least n nodes, one node for each variable in the argument. The proof strategy follows a similar structure to the *BDD Claim* proof in Thm. 3.3. We prove the claim by contradiction. Suppose that there is some BDD B for H_n that does not need at least one node for each variable. In that case, the H_n function represented by B does not depend on that particular variable. Let $\mathcal{T}$ denote the "all-true" assignment of variables; i.e., $\mathcal{T} \stackrel{\text{def}}{=} \forall k \in \{0..n/2-1\}, x_k \mapsto T, y_k \mapsto T$. There are three possible situations:

- Case 1: B does not depend on variable y_k, for some $k \in \{0..n/2-1\}$. Consider two variable assignments: $A_1 \stackrel{\text{def}}{=} \mathcal{T}$ and $A_2 \stackrel{\text{def}}{=} \mathcal{T}[y_k \mapsto F]$ (i.e., A_2 is A_1 with y_k updated to F).

  A_1 and A_2 yield the same value for the function represented by B, but they yield different values for the Hadamard relation. That is, if $H_n[A1] = v$ (where v is either 1 or -1), then $H_n[A2] = -v$. We prove this claim by induction on level.

  Proof. Base Case:

  - $n = 2$. $H_2[A_1]$ is the lower-right corner of Fig. 2.2a, which is -1, and $H_2[A_2]$ is the value of the path $[x_0 \mapsto T, y_0 \mapsto F]$, which yields 1.
  - $n = 4$. A_1 is the path to the rightmost (16th) leaf in Fig. 2.2c, which yields a value of 1. For $k = 0$, A_2 ends up at the 12th leaf, which is -1; if $k = 1$, A_2 ends up at the 15th leaf, which is also -1.

  Induction Step: Let us extend the notation for A_1 and A_2 by adding level information. A_1^m denotes the "all-true" assignment for 2^m variables, and $A_2^m = A_1^m[y_k \mapsto F]$. Let us assume that the claim is true for H_{2^m}, i.e., $H_{2^m}[A_1] = v$ (could be 1 or -1) and $H_{2^m}[A_2] = -v$. We must show that the claim holds true for $H_{2^{m+1}}$.

  We know that $H_{2^{m+1}} = H_{2^m} \otimes H_{2^m}$. Thus, $H_{2^{m+1}}[A_1^{m+1}] = H_{2^m}[A_1^m] * H_{2^m}[A_1^m]$, where $A_1^{m+1} = A_1^m || A_1^m$. Thus, $H_{2^{m+1}}[A_1^{m+1}]$ must have the value v^2 $(= v * v)$. A recursive relation can similarly be written for assignment A_2, depending on where the bit-flip for y occurs. There are two possible cases:

  1. k occurs in the first half; $A_2^{m+1} = A_2^m || A_1^m$ and therefore, $H_{2^{m+1}}[A_2^{m+1}] = H_{2^m}[A_2^m] * H_{2^m}[A_1^m]$, which leads to a value of $-v^2$ $(= -v * v)$.
  2. k occurs in the second half; $A_2^{m+1} = A_1^m || A_2^m$ and therefore, $H_{2^{m+1}}[A_2^{m+1}] = H_{2^m}[A_1^m] * H_{2^m}[A_2^m]$ which leads to a value of $-v^2$ $(= v * -v)$.

In both cases, the values obtained with a bit-flip do not match the value for an "all-true" assignment. □

We conclude that none of the y_k variables can be dropped individually.

- Case 2: None of the x variables can be dropped individually, using a completely analogous argument to Case 1.
- Case 3: B does not depend on either x_k or y_k. The assignments $A_1 = \mathcal{T} = [..., x_k \mapsto T, y_k \mapsto T, ...]$, $A_2 = \mathcal{T}[y_k \mapsto F] = [..., x_k \mapsto T, y_k \mapsto F, ...]$, $A_3 = \mathcal{T}[x_k \mapsto F] = [..., x_k \mapsto F, y_k \mapsto T, ...]$ and $A_4 = \mathcal{T}[x_k \mapsto F, y_k \mapsto F] = [..., x_k \mapsto F, y_k \mapsto F, ...]$ must be mapped to different values by the function represented by B, which violates the definition of the Hadamard relation H_n. (More precisely, for $n \geq 4$, $H_n[A_1] = 1$, but $H_n[A_2] = H_n[A_3] = H_n[A_4] = -1$, which can be proved using an inductive argument similar to Case 1.) Consequently, (x_k, y_k) cannot be dropped as a pair.

Because (i) no y_k can be dropped individually, (ii) no x_k can be dropped individually and (iii) no (x_k, y_k) pair can be dropped together, B—and hence any BDD representation for H_n, requires $\Omega(n)$ nodes. □

3.7.3 The Addition Relation ADD_n

Definition 3.5. The addition relation $ADD_n : \{0,1\}^{n/3} \times \{0,1\}^{n/3} \times \{0,1\}^{n/3} \to \{0,1\}$ on variables $(x_0 \cdots x_{n/3-1})$, $(y_0 \cdots y_{n/3-1})$, and $(z_0 \cdots z_{n/3-1})$ is the relation $ADD_n(X,Y,Z) \overset{\text{def}}{=} Z = (X + Y \mod 2^{n/3})$.

Theorem 3.5 Exponential separation for the addition relation**.** For $n = 3 \cdot 2^l$, where $l \geq 0$, ADD_n can be represented by a CFLOBDD with $\mathcal{O}(\log n)$ vertices and edges. In contrast, a BDD that represents ADD_n requires $\Omega(n)$ nodes.

Proof.

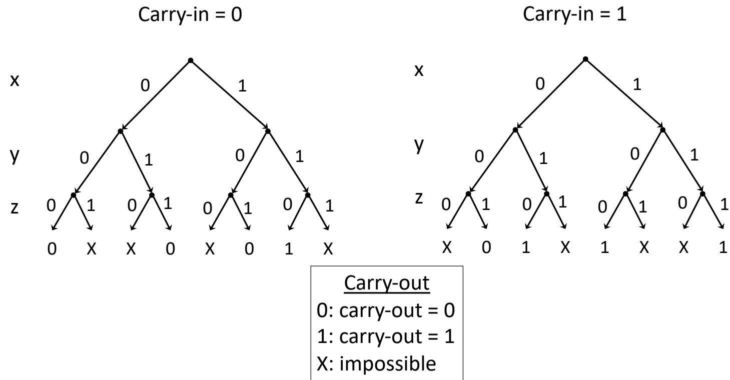


Figure 3.13: Decision diagrams for the *carry-in* = 0 and *carry-in* = 1 cases of ADD_n

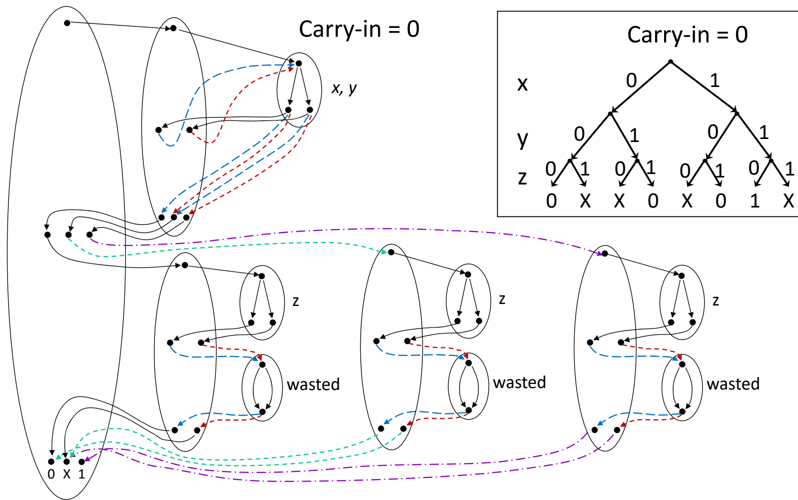


Figure 3.14: CFLOBDD representation for the *carry-in* = 0 case of ADD_n

CFLOBDD Claim. For a given bit position i, the representation has to distinguish between two cases for the carry-bit value coming from bit position $i-1$: *carry-in* $= 0$ and *carry-in* $= 1$. Fig. 3.13 shows decision trees for the *carry-in* $= 0$ and *carry-in* $= 1$ cases. The terminal values 0 and 1 indicate the carry-out value to be passed to bit position $i+1$. Terminal value X represents a failure case: with the given carry-in value c_i, and the given values for x_i, y_i, and z_i, $z_i \neq c_i \oplus x_i \oplus y_1$.

The CFLOBDD representation for ADD_n using the interleaved-variable ordering is shown in Figs. 3.14–3.17. Our claim is that ADD_n has $\mathcal{O}(\log n)$ vertices and

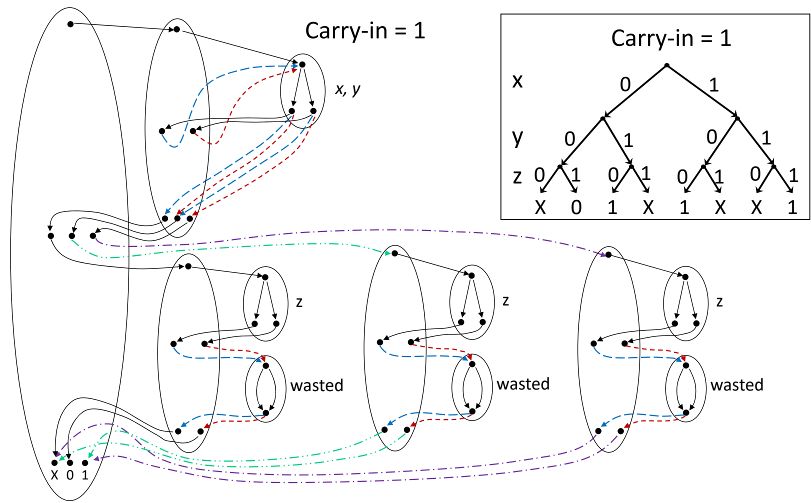


Figure 3.15: CFLOBDD representation for the *carry-in* = 1 case of ADD_n

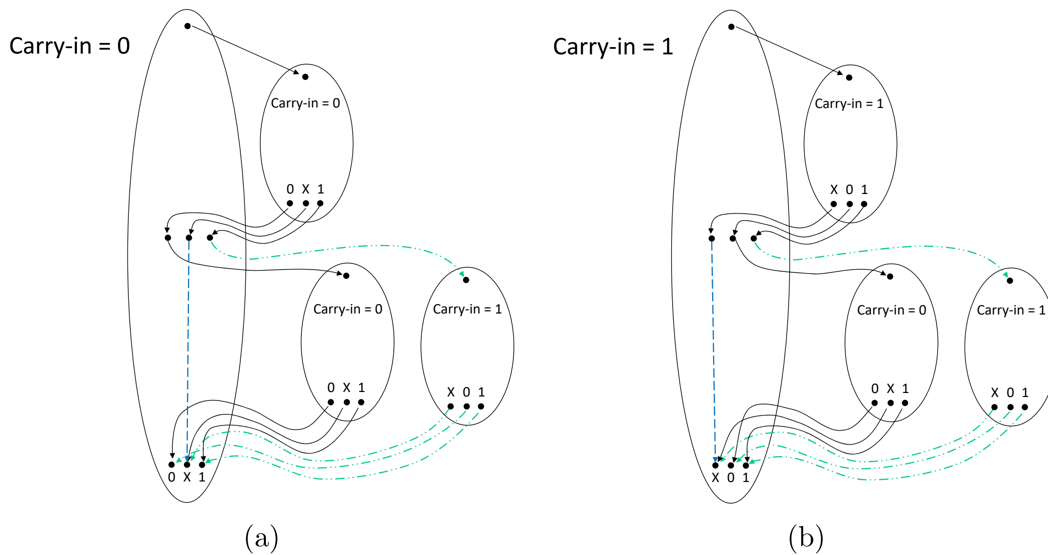


Figure 3.16: Recursive CFLOBDD structures for the (a) *carry-in* = 0 and (b) *carry-in* = 1 cases of ADD_n. To reduce clutter, a B-connection to a NoDistinctionProtoCFLOBDD is depicted as a straight dashed line to the X exit vertex.

edges. To keep triples of variables x_i, y_i, and z_i aligned with the power-of-two nature of CFLOBDDs, our representation of ADD_n uses an additional set of $n/3$ dummy variables (which will always be "routed" through a DontCareGrouping, so they have

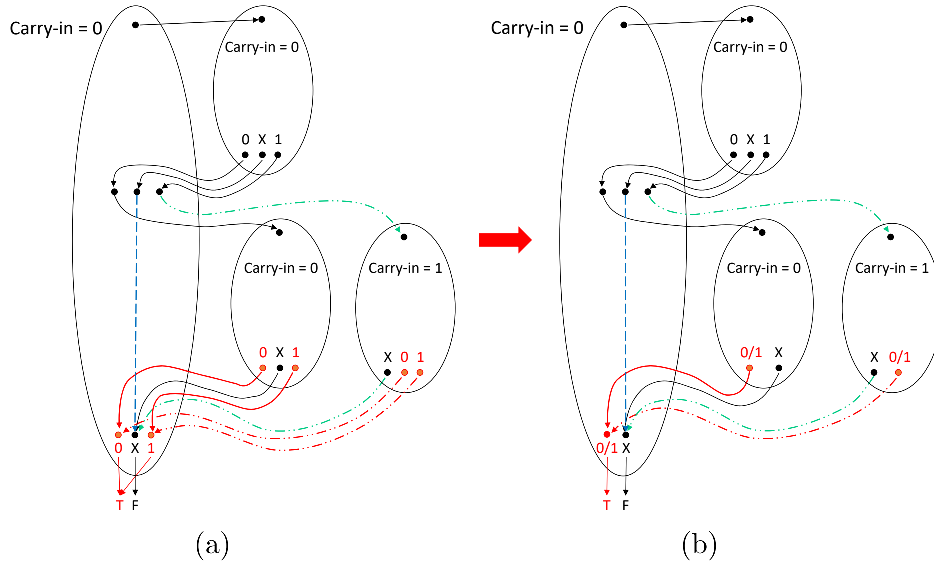


Figure 3.17: (a) The recursive CFLOBDD structure of the ADD_n relation. Because the 0 and 1 exit vertices of the top-level grouping are associated with the single terminal T (while X is mapped to F), the 0 and 1 exits must be coalesced (indicated in red), which propagates to the internal groupings of the CFLOBDD. (b) The propagation of coalesced exit vertices.

no effect on the relation over X, Y, and Z).

Fig. 3.14 shows the proto-CFLOBDD at level 2 for the *carry-in* $= 0$ case for some triple of variables x_i, y_i, and z_i in the ADD_n CFLOBDD. The DontCareGroupings labeled "wasted" in Fig. 3.14 are for the associated i^{th} dummy variable. Fig. 3.14 is one of two "leaf" building blocks that handles a triple of variables; the other is shown in Fig. 3.15, which depicts the CFLOBDD representation for the *carry-in* $= 1$ case for x_i, y_i, and z_i.

Fig. 3.16 shows the two recursive structures used at all higher levels (¿ 2) of the CFLOBDD for the *carry-in* $= 0$ and *carry-in* $= 1$ cases. Note that at all levels, including the base case of Fig. 3.14, the sequence for the exit vertices of *carry-in* $= 0$ proto-CFLOBDDs is $[0, X, 1]$. Similarly, at all levels the sequence for the exit vertices of *carry-in* $= 1$ proto-CFLOBDDs is $[X, 0, 1]$. Consequently, at each level, we can construct the *carry-in* $= 0$ and *carry-in* $= 1$ proto-CFLOBDDs by adding just one

grouping for each case, and each of the groupings contains a fixed number of vertices and edges.

At the outermost level, we use the *carry-in* $= 0$ proto-CFLOBDD (which has an A-connection to the *carry-in* $= 0$ proto-CFLOBDD at the next lower level, but B-connections to both the *carry-in* $= 0$ and *carry-in* $= 1$ variants). The exit vertices labeled 0 and 1 are connected to terminal value T, and exit vertex X to terminal value F. Fig. 3.17a shows this structure.

To obey structural invariant 6 of Defn. 3.3, it is necessary to collapse the 0 and 1 exit vertices at the outermost level, because both lead to the terminal value T. This collapsing process propagates to the different levels of internal groupings of the CFLOBDD. What remains to be established is that the collapsing step does not cause the CFLOBDD as a whole to blow up in size.[16]

Fortunately, as indicated in Fig. 3.17b, the propagation only takes place in groupings in B-connections (and B-connections of B-connections, etc.), and does not propagate to A-Connection groupings. As we prove below, the collapsing step only produces two more kinds of proto-CFLOBDDs at each level: (i) a *carry-in* $= 0$ variant in which the exit-vertex sequence is $[0/1, X]$, and (ii) a *carry-in* $= 1$ variant in which the exit-vertex sequence is $[X, 0/1]$ (where $0/1$ denotes the coalescing of the 0 and 1 exit vertices). Because there are now at most four grouping patterns that arise at each level, the final CFLOBDD at most doubles in size. The overall size of the CFLOBDD is proportional to the outermost level, and hence the CFLOBDD representation of ADD_n achieves double-exponential compression with respect to the size of the decision tree for ADD_n (i.e., $\mathcal{O}(\log n)$ in vertices and edges, and $\mathcal{O}(l)$ in the number of levels ($l = \log n$)).

More formally, let $R(l)$ and $P(l)$ denote the number of vertices and edges at or below level l in the proto-CFLOBDD representation of ADD_n for *carry-in* $= 0$ and

[16]The reason a “collapsing” step could cause a size blow-up is because occurrences of previously shared identical substructures could turn into multiple substructures, each slightly different.

carry-in = 1, respectively, *without* collapsing the 0 and 1 exits. Also, let $A(l)$ and $B(l)$ denote the number of vertices and edges at or below level l in the proto-CFLOBDD representation of ADD_n for *carry-in* = 0 and *carry-in* = 1, respectively, *with* 0 and 1 exits collapsed. We can give mutually recursive recurrence equations for $R(l)$ and $P(l)$. For $R(l)$, we have

$$R(l) = R(l-1) + P(l-1) + 7 + 13, \tag{3.14}$$

which defines $R(l)$ in terms of $R(l-1)$ and $P(l-1)$ according to the pattern given in Fig. 3.16a. Although Fig. 3.16a shows two *carry-in* = 0 proto-CFLOBDDs at level $l-1$, they are actually shared data structures, and so $R(l-1)$ only counts once in Eqn. (3.14). We call this property the "*same-structure sharing*" property: stated another way, all non-zero coefficients of R, P, A, and B terms can be replaced by 1. The 7 in Eqn. (3.14) represents the number of vertices at level l, and 13 refers to the edge count between groupings at level l and level $l-1$. Similarly, the recurrence equation for $P(l)$, following the pattern in Fig. 3.16b, is

$$P(l) = P(l-1) + R(l-1) + 7 + 13. \tag{3.15}$$

Combining Eqns. (3.14) and (3.15), we obtain

$$\begin{aligned}
R(l) + P(l) &= (R(l-1) + P(l-1) + 7 + 13) + (P(l-1) + R(l-1) + 7 + 13) \\
&= 2R(l-1) + 2P(l-1) + 2 * (7 + 13) && \text{Collecting terms} \\
&= R(l-1) + P(l-1) + 40 && \text{Same-structure sharing} \\
&= R(l-2) + P(l-2) + 2 * 40 && \text{Substitution} \\
&\quad \vdots \\
&= R(l-k) + P(l-k) + k * 40 && \text{For a general } k \\
&\quad \vdots \\
&= R(2) + P(2) + (l-2) * 40 \\
&= \mathcal{O}(1) + \mathcal{O}(l) && \text{From Figs. 3.14 and 3.15} \\
&= \mathcal{O}(l) && (3.16)
\end{aligned}$$

Substituting Eqn. (Collecting terms) in Eqns. (3.14) and (3.15), we obtain

$$\begin{aligned} R(l) &= \mathcal{O}(l-1) + 20 \\ &= \mathcal{O}(l-1) + \mathcal{O}(1) \\ &= \mathcal{O}(l) \end{aligned} \tag{3.17}$$

$$\begin{aligned} P(l) &= \mathcal{O}(l-1) + 20 \\ &= \mathcal{O}(l-1) + \mathcal{O}(1) \\ &= \mathcal{O}(l) \end{aligned} \tag{3.18}$$

The argument for $A(l)$ and $B(l)$ is similar. We define $A(l)$ in terms of $R(l-1)$, $A(l-1)$, and $B(l-1)$ following the pattern in Fig. 3.17b.

$$A(l) = R(l-1) + A(l-1) + B(l-1) + 6 + 12, \tag{3.19}$$

where the numbers of vertices and edges added at each level are 6 and 12, respectively. Similarly, the recurrence equation for $B(l)$ is

$$B(l) = P(l-1) + A(l-1) + B(l-1) + 6 + 12 \tag{3.20}$$

Combining Eqns. (3.19) and (3.20), we obtain

$$
\begin{aligned}
A(l) + B(l) &= (R(l-1) + A(l-1) + B(l-1) + 6 + 12) && (3.21)\\
&+ (P(l-1) + A(l-1) + B(l-1) + 6 + 12) \\
&= R(l-1) + P(l-1) + 2A(l-1) + 2B(l-1) + 2 * (6 + 12) && (3.22)\\
& && \text{Collecting terms}\\
&= R(l-1) + P(l-1) + A(l-1) + B(l-1) + 36 && (3.23)\\
& && \text{Same-structure sharing}\\
&= (R(l-2) + P(l-2) + 20) + (P(l-2) + R(l-2) + 20) && (3.24)\\
&+ (R(l-2) + A(l-2) + B(l-2) + 6 + 12) && (3.25)\\
&+ (P(l-2) + A(l-2) + B(l-2) + 6 + 12) + 36 && \text{Substitution}\\
&= 3R(l-2) + 3P(l-2) + 2A(l-2) + 2B(l-2) + 40 + 36 + 36 && (3.26)\\
& && \text{Collecting terms}\\
&= R(l-2) + P(l-2) + A(l-2) + B(l-2) + 40 + 2 * 36 && (3.27)\\
& && \text{Same-structure sharing}\\
&\vdots \\
&= R(l-k) + P(l-k) + A(l-k) + B(l-k) + (k-1) * 40 + k * 36 && (3.28)\\
& && \text{For a general } k\\
&\vdots \\
&= R(2) + P(2) + A(2) + B(2) + (l-3) * 40 + (l-2) * 36 \\
&= \mathcal{O}(1) + \mathcal{O}(l) && \text{From Figs. 3.14 and 3.15}\\
&= \mathcal{O}(l) && (3.29)
\end{aligned}
$$

Using Eqn. (3.21), we can rewrite $A(l)$ and $B(l)$ as follows:

$$
\begin{aligned}
A(l) &= \mathcal{O}(l-1) + \mathcal{O}(l-1) + 18 \\
&= \mathcal{O}(l-1) + \mathcal{O}(1) && (3.30)\\
&= \mathcal{O}(l)
\end{aligned}
$$

$$
\begin{aligned}
B(l) &= \mathcal{O}(l-1) + \mathcal{O}(l-1) + 18 \\
&= \mathcal{O}(l-1) + \mathcal{O}(1) \\
&= \mathcal{O}(l)
\end{aligned}
\tag{3.31}
$$

Eqns. (3.17), (3.18), (3.30), and (3.31) show that $R(l)$, $P(l)$, $A(l)$, and $B(l)$ are all linear in the number of levels—$\mathcal{O}(l)$—and logarithmic in the number of vertices and edges—$\mathcal{O}(\log n)$. Thus, for the interleaved-variable ordering for (vectors of) variables X, Y, and Z, the vertices and edges count for the CFLOBDD for ADD_n is $\mathcal{O}(\log n)$.

BDD Claim. To represent the addition relation ADD_n as a BDD, we claim that we require at least n nodes, one node for each variable in the argument, regardless of the variable ordering.

We will prove this claim by contradiction, similar to the *BDD Claim* proofs for EQ_n and H_n. We assume that a BDD representation B of ADD_n does not need at least one node for each variable, and therefore the BDD representation of ADD_n does not depend on that particular variable. Let us define $\mathcal{F}$ as an "all-false" assignment of variables, i.e., $\mathcal{F} \stackrel{\text{def}}{=} \forall k \in \{0...n/3-1\}, x_k \mapsto F, y_k \mapsto F, z_k \mapsto F$. There are seven possible cases:

- Case 1: B does not depend on variable y_k, for some $k \in \{0..n/3-1\}$. Let us consider two different assignments of variables: $A_1 \stackrel{\text{def}}{=} \mathcal{F}$ and $A_2 \stackrel{\text{def}}{=} \mathcal{F}[y_k \mapsto T]$; i.e., A_2 is A_1 with y_k updated to T. (Note that $A_1 = [..., x_k \mapsto F, y_k \mapsto F, z_k \mapsto F, ...]$ and $A_2 = [..., x_k \mapsto F, y_k \mapsto T, z_k \mapsto F, ...]$.) Because B does not depend on y_k, $B[A_1]$ must equal $B[A_2]$, which violates the definition of addition relation ADD_n; in particular, $ADD_n[A_1] = 1$ and $ADD_n[A_2] = 0$. Let us show how the violation occurs by using the example of $n = 24$ and $k = 3$ ($\in \{0..7\}$). We have $ADD_n[A_1] = 1$ because the following triple of numbers is a correct instance of an addition problem:

$$
\begin{array}{r}
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
+\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
\hline
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0
\end{array}
$$

However, $ADD_n[A_2] = 0$ because the following triple is an incorrect instance of addition:

$$\begin{array}{ccccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

We conclude that none of the y_k variables can be dropped individually. The argument for arbitrary n and k is completely analogous.

- Case 2: B does not depend on the pair (x_k, y_k). We use a similar proof strategy as Case 1. Consider two assignments $A_1 \stackrel{\text{def}}{=} \mathcal{F}$ and $A_2 \stackrel{\text{def}}{=\!=} A_1[x_k \mapsto T][y_k \mapsto T]$. The assignments must produce the same value for the function represented by B, but they yield different values for the ADD_n relation. Again, let us show how the violation occurs by using the example of $n = 24$ and $k = 3$. We have $ADD_n[A_1] = 1$ because the following triple is a correct instance of an addition problem:

$$\begin{array}{ccccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ + & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

  However, $ADD_n[A_2] = 0$ because the following triple is an incorrect instance of addition:

$$\begin{array}{ccccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

  We conclude that (x_k, y_k) cannot be dropped as a pair.

We can make arguments similar to the ones above for other combinations of variables, such as (i) B does not depend on variable x_k individually, (ii) B does not depend on variable z_k individually, (iv) B does not depend on either y_k or z_k variables as a pair, (v) B does not depend on either x_k or z_k variables as a pair, (vi) B does not depend on all three variables x_k, y_k, z_k together. Consequently, we conclude that B—and hence any BDD representation for ADD_n requires $\Omega(n)$ nodes. □

3.8 Applications to Quantum Algorithms

In this section, we will discuss the application of CFLOBDDs to quantum simulation. For more details on quantum simulation in general, refer to §2.3.

A quantum state could be encoded with a decision tree of height n, but such a representation would be inefficient. The potential of CFLOBDDs is for providing (up to) double-exponential compression in the sizes of the vectors and matrices that arise during quantum simulation, using $\log n$ and $\log n + 1$ levels, respectively. Because many quantum gates can be described using Kronecker products, there is great potential for them to have a compact representation as a CFLOBDD.

The evaluation of CFLOBDDs for quantum simulation in §3.9.2.2 uses the CFLOBDD representations of gate matrices and state vectors presented in this section, namely, multi-terminal CFLOBDDs with a semiring value at each terminal value (à la ADDs Bahar et al. (1997)). To support functions of type $\{0,1\}^n \to \mathbb{C}$, we implemented a semiring of multi-precision-floating-point Fousse et al. (2007) complex numbers.

3.8.1 Special Matrices

In this section, we discuss how matrices and vectors used in quantum algorithms can be efficiently represented using CFLOBDDs.

3.8.1.1 Hadamard Gate

As we saw in §3.2, a Hadamard matrix can be efficiently represented by a CFLOBDD. Hadamard matrices can be recursively defined as $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$; thus, the Hadamard matrix at level $l+1$ can be constructed using a Kronecker product of level-l Kronecker-product matrices (§3.6.5). Alternatively, it is possible to bypass such Kronecker-product operations and directly construct the Hadamard matrix for a given level. Pseudo-code for the latter approach is given as Alg. 26. The algorithm takes as input the max-level l, and returns the CFLOBDD of level l that

Algorithm 26: Hadamard Matrix

```
1  Algorithm HadamardMatrixCFLOBDD(l)
     Input: int l – level of the CFLOBDD to be constructed; #variables
            = 2^l
     Output: The CFLOBDD that represents H_{2^l}, of size 2^{2^{l-1}} × 2^{2^{l-1}}
2    begin
3      Grouping g = HadamardMatrixGrouping(l);
4      return RepresentativeCFLOBDD(g, [1,-1]);
5    end
6  end
1  SubRoutine HadamardMatrixGrouping(l)
     Input: int l – the level of the proto-CFLOBDD to be constructed
     Output: Grouping g representing a proto-CFLOBDD for H_{2^l}
2    begin
3      InternalGrouping g = new InternalGrouping(l);
4      if i == 1 then
5        g.AConnection = ForkGrouping;
6        g.AReturnTuple = [1,2];
7        g.numberOfBConnections = 2;
8        g.BConnection[1] = DontCareGrouping;
9        g.ReturnTuples[1] = [1];
10       g.BConnection[2] = ForkGrouping;
11       g.BReturnTuples[2] = [1,2];
12     else
13       Grouping g' = HadamardMatrixGrouping(l-1);
14       g.AConnection = g';
15       g.AReturnTuple = [1,2];
16       g.numberOfBConnections = 2;
17       g.BConnection[1] = g';
18       g.BReturnTuples[1] = [1,2];
19       g.BConnection[2] = g';
20       g.BReturnTuples[2] = [2,1];
21     end
22     g.numberOfExits = 2;
23     return RepresentativeGrouping(g);
24   end
25 end
```

represents the square matrix H_{2^l} of size $2^{2^{l-1}} \times 2^{2^{l-1}}$. The base case, for $l = 1$, returns the representation of

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

In Alg. 26, line [4], the value tuple is always $[1, -1]$ (and never $[-1, 1]$) because the $[0, 0]$ entry of every Hadamard matrix is 1.

As noted in footnote 3, a Hadamard gate is a Hadamard matrix scaled by a power of $\frac{1}{\sqrt{2}}$. In particular, the scale factor for H_{2^l} is $\left(\frac{1}{\sqrt{2}}\right)^l$. In our summaries of quantum algorithms in §3.8.2, we omit such scale factors to reduce clutter.

3.8.1.2 Identity Gate

The identity matrix, which is the same as the equality relation EQ_N, can be efficiently represented by a CFLOBDD, as shown in §3.7.1. The identity matrix at level $l + 1$ can be recursively computed from identity matrices at level l as $I_{2^{l+1}} = I_{2^l} \otimes I_{2^l}$. The CFLOBDD for the identity matrix at level $l{+}1$ can either be constructed using a Kronecker product of level-l identity-matrix CFLOBDDs (§3.6.5), or it can be constructed directly. Pseudo-code for the latter approach is given as Alg. 27. The algorithm takes as input the max-level l, and returns the CFLOBDD of level l that represents I_{2^l} (of size $2^{2^{l-1}} \times 2^{2^{l-1}}$). The base case, for $l = 1$, returns the representation of

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

3.8.1.3 Not Gate

The not matrix, denoted by X_2, flips the elements of the vector to which it is applied. X_2 is defined as follows:

$$X_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

X_2 is similar to I_2, but with 0 and 1 swapped. Hence, the representation of X_2 uses the same level-1 proto-CFLOBDD used in I_2, but has terminal values $[0, 1]$ (whereas

Algorithm 27: Identity Matrix

```
1  Algorithm IdentityMatrixCFLOBDD(l)
     Input: int l – level of the CFLOBDD to be constructed; #variables
            = 2^l
     Output: The CFLOBDD that represents I_{2^{2^{l-1}}}, of size 2^{2^{l-1}} × 2^{2^{l-1}}
2    begin
3      Grouping g = IdentityMatrixGrouping(l);
4      return RepresentativeCFLOBDD(g, [1,0]);
5    end
6  end
1  SubRoutine IdentityMatrixGrouping(l)
     Input: int l – the level of the proto-CFLOBDD to be constructed
     Output: Grouping g representing a proto-CFLOBDD for I_{2^{2^{l-1}}}
2    begin
3      InternalGrouping g = new InternalGrouping(l);
4      if i == 1 then
5        g.AConnection = ForkGrouping;
6        g.AReturnTuple = [1,2];
7        g.numberOfBConnections = 2;
8        g.BConnection[1] = ForkGrouping;
9        g.ReturnTuples[1] = [1,2];
10       g.BConnection[2] = ForkGrouping;
11       g.BReturnTuples[2] = [2,1];
12     else
13       Grouping g' = IdentityMatrixGrouping(l-1);
14       g.AConnection = g';
15       g.AReturnTuple = [1,2];
16       g.numberOfBConnections = 2;
17       g.BConnection[1] = g';
18       g.BReturnTuples[1] = [1,2];
19       g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
20       g.BReturnTuples[2] = [2];
21     end
22     g.numberOfExits = 2;
23     return RepresentativeGrouping(g);
24   end
25 end
```

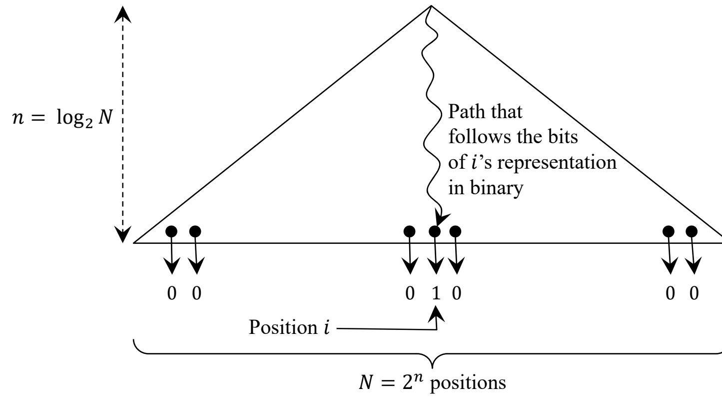


Figure 3.18: One-hot vector as the yield of a decision tree. The single occurrence of 1 is at the leaf indexed by i—i.e., at the end of the path from the root that follows the bits of i's representation in binary.

Algorithm 28: Standard Basis Vector

```
1 Algorithm StandardBasisVectorCFLOBDD(l, i)
    Input: int l - level of the CFLOBDD = log n, where n = number of
           bits; int i - index
    Output: The CFLOBDD that represents e_x where x = the binary
            representation of i in n bits
2   begin
3       Grouping g = StandardBasisVectorGrouping(l);
4       ValueTuple valueTuple = (i == 0) ? [1,0] : [0,1] // 1^st elem. is
         0 unless i is 0;
5       return RepresentativeCFLOBDD(g, valueTuple);
6   end
7 end
```

I_2 has $[1, 0]$).

3.8.1.4 Standard-Basis Vectors

The family of standard-basis vectors are an important set of vectors in quantum algorithms because they represent the basis states. A standard-basis vector is a one-hot vector—a vector all of whose elements are 0, except for a single 1. In the terminology of formal-language theory, the vector to picture is the *yield* of the deci-

Algorithm 29: Standard Basis Vector (cont.)

```
1  SubRoutine StandardBasisVectorGrouping(l, i)
     Input: int l - level of the CFLOBDD = log n, where n = number of
            bits; int i - index
     Output: Grouping g that represents e_x where x = the binary
             representation of i in n bits. (Exit vertex 2 corresponds to
             e_x unless x = 0...0, in which case exit vertex 1 corresponds
             to e_x.)
2    begin
3      if l == 0 then
4        return RepresentativeForkGrouping;
5      end
6      InternalGrouping g = new InternalGrouping(l);
7      int higherOrderIndex = i >> (1 << (l - 1)) // First half of x;
8      g.AConnection = StandardBasisVectorGrouping(l-1,
        higherOrderIndex);
9      g.AReturnTuple = [1,2];
10     g.numberOfBConnections = 2;
11     int lowerOrderIndex = i & ((1 << (1 << (l - 1))) - 1)  // Second
        half of x;
12     Grouping g' = StandardBasisVectorGrouping(l-1,
        lowerOrderIndex);
13     if higherOrderIndex == 0 then
14       g.BConnection[1] = g'  // Connection 1 = on path for x;
15       g.BReturnTuples[1] = [1,2];
16       g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
17       g.BReturnTuples[2] = (lowerOrderIndex == 0) ? [2] : [1];
18     else
19       g.BConnection[1] = NoDistinctionProtoCFLOBDD(l-1);
20       g.BReturnTuples[1] = [1];
21       g.BConnection[2] = g'  // Connection 2 = on path for x;
22       g.BReturnTuples[2] = (lowerOrderIndex == 0) ? [2,1] : [1,2];
23     end
24     g.numberOfExits = 2;
25     return RepresentativeGrouping(g);
26   end
27 end
```

sion tree obtained by unfolding the CFLOBDD (see Fig. 3.18). Let a standard-basis vector of size $2^n \times 1$ with its single occurrence of 1 at position i be denoted by e_x, where x is the binary representation of i using n bits. (The vector $e_{0\ldots0}$ is the initial state in many quantum algorithms.)

A representation as a CFLOBDD of a standard-basis vector for one-hot position i $(= x)$ can be created in n steps via the pseudo-code given as Algs. 28 and 29. The code is relatively straightforward, with the small complication that $x = 0\ldots0$ is a special case that has to be accounted for at every level of recursion of auxiliary function StandardBasisVectorGrouping (as the bit-string x becomes of half-size, quarter-size, etc.—see lines [7]–[8] and lines [11]–[12]). The invariant for procedure StandardBasisVectorGrouping on input i $(= x)$ is that exit vertex 2 always corresponds to e_x—unless $x = 0\ldots0$, in which case exit vertex 1 corresponds to e_x.

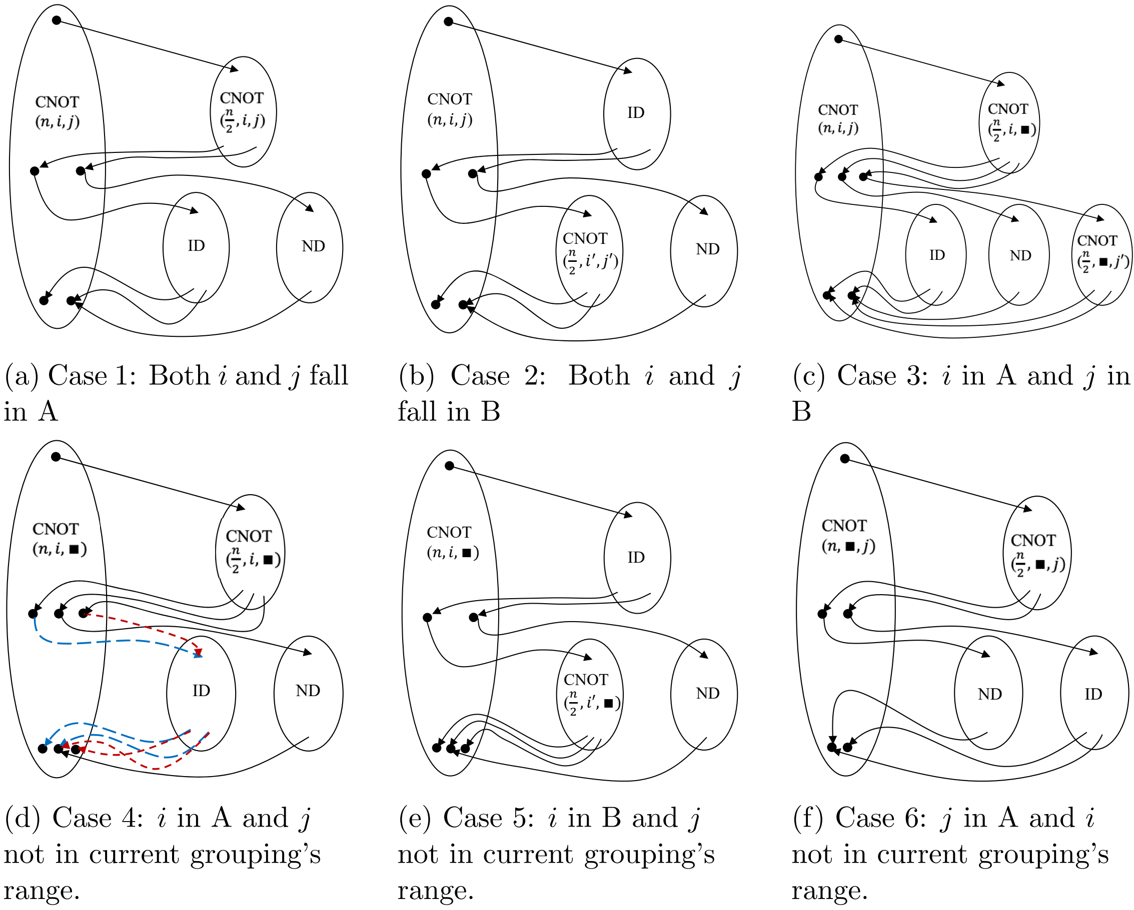
The base case is for a 2×1 vector, either $e_0 = [1,0]^t$ or $e_1 = [0,1]^t$, depending on the current value of x. e_0 would be represented by a ForkGrouping (with distinguished exit vertex 1, because $x = 0$); e_1 would be represented by a ForkGrouping (with distinguished exit vertex 2, because $x = 1$).

3.8.1.5 Controlled-NOT Gate

A Controlled-NOT (CNOT) is an operation involving two index bits: one bit is the control-bit and the other is the controlled-bit; in the output, the value of the controlled-bit is flipped if the control-bit is '1'. The matrix that implements this behavior for two bits, denoted by $CNOT_2$, where the first bit is the control-bit and the second bit is the controlled-bit, is as follows:

$$CNOT_2 = \begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} = \begin{bmatrix} I_2 & O_2 \\ O_2 & X_2 \end{bmatrix} \tag{3.32}$$

Note that when the first bit is 1 and the second bit is flipped, the 2×2 block in the lower right is the not matrix X_2. (O_2 denotes the 2×2 matrix of zeros.)



(a) Case 1: Both i and j fall in A

(b) Case 2: Both i and j fall in B

(c) Case 3: i in A and j in B

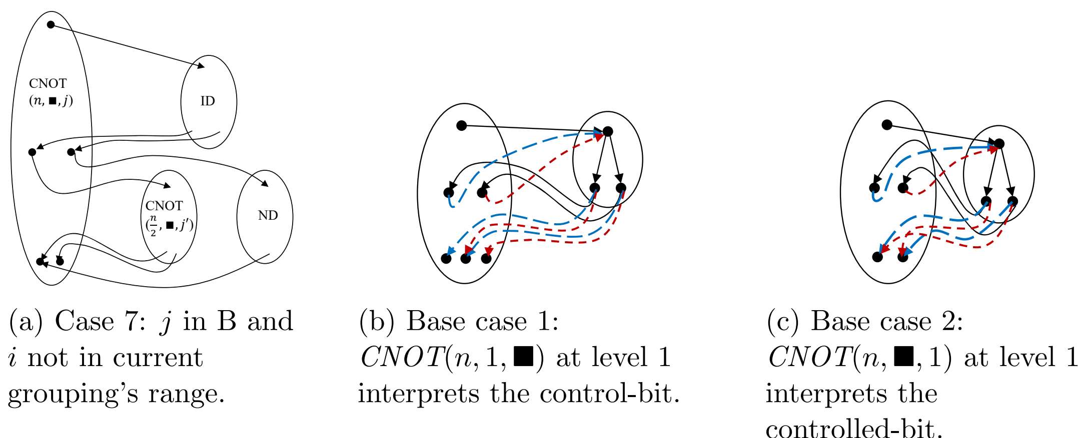
(d) Case 4: i in A and j not in current grouping's range.

(e) Case 5: i in B and j not in current grouping's range.

(f) Case 6: j in A and i not in current grouping's range.

Figure 3.19: The different cases of the CNOT construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`; ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). CNOT takes 3 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, and j for the controlled-bit, where $0 \leq i < j < n$. i' and j' denote bit indices adjusted according to the level l: $i' = i - 2^{l-1}$; $j' = j - 2^{l-1}$. A black square indicates that a particular index is outside the grouping's index range.

For larger numbers of bits, the situation is more complex. With four bits, and the second bit controlling the third, we have



(a) Case 7: j in B and i not in current grouping's range.

(b) Base case 1: $CNOT(n, 1, \blacksquare)$ at level 1 interprets the control-bit.

(c) Base case 2: $CNOT(n, \blacksquare, 1)$ at level 1 interprets the controlled-bit.

Figure 3.20: The different cases of the CNOT construction, continued. Figures (b) and (c) show the two base cases at level 1, for $CNOT(n, 1, \blacksquare)$ and $CNOT(n, \blacksquare, 1)$, respectively.

$$CNOT(4,2,3) = \begin{bmatrix} I_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & I_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & I_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & I_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & I_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & I_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & I_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & I_2 & O_2 \end{bmatrix} = I_2 \otimes CNOT_2 \otimes I_2.$$

However, with four bits and the first bit controlling the third, we have

$$CNOT(4,1,3) = \begin{bmatrix} I_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & I_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & I_2 & O_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & I_2 & O_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & X_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & X_2 & O_2 & O_2 & O_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & X_2 \\ O_2 & O_2 & O_2 & O_2 & O_2 & O_2 & X_2 & O_2 \end{bmatrix}$$

for which there is no clean expression in terms of $\otimes$.

Fortunately, we can create a double-exponentially compressed CFLOBDD representation of $CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$. Suppose that the control-bit is bit i,

and the controlled-bit is bit j; then for a grouping g, there are eight cases to consider (here we discuss the cases for $i < j$), [17] which are depicted in Figs. 3.19 and 3.20. The key to understanding Figs. 3.19 and 3.20 is that the construction maintains the invariant shown in Eqn. (3.33) on the exit vertices of the different kinds of groupings.

	Role	Significance of exit vertex		
		1	2	3
Proto-CFLOBDD	$CNOT(n, i, j)$	on-path	off-path	N/A
	$CNOT(n, i, \blacksquare)$	on-path	off-path	Controlled-bit flipped
	$CNOT(n, \blacksquare, j)$	off-path	on-path	N/A
	ID	on-path	off-path	N/A
$CFLOBDD$	Top level	1	0	N/A

(3.33)

Here, “on-path” means that the exit occurs on a matched-path that can be continued to the top-level terminal value 1; “off-path” means that it will only be used to reach the top-level terminal value 0. For instance, in Fig. 3.19(a)–(f), and Fig. 3.20(a), each occurrence of ND (a NoDistinctionProtoCFLOBDD) is attached to the middle vertex of a grouping g that is reached from an A-connection’s off-path exit. The NoDistinctionProtoCFLOBDD has one exit vertex for which the return edge connects it to the off-path exit for g.

1. Both i and j fall in the A-connection of g. Fig. 3.19(a) shows the structural representation for this scenario. The bits in g’s A-connection handle the CNOT operation. The bits in g’s B-connection are not involved, and hence the “on-path” middle vertex is connected to Identity, and the “off-path” middle vertex is connected to a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix).

2. Both i and j fall in g’s B-connection range. This scenario is depicted in Fig. 3.19(b). The bits in g’s A-connection do not affect the CNOT operation,

[17] The general case—$CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$, where j is allowed to be less than i—would be similar, but with some additional cases.

and hence we have an Identity matrix in the A-connection. The B-connection handles the relationship between the control-bit and controlled-bit through a CNOT structure of the form $CNOT(\frac{n}{2}, i', j')$. i' and j' denote bit indices adjusted according to the level l: $i' = i - 2^{l-1}$; $j' = j - 2^{l-1}$.

3. i lies in g's A-connection range and j in g's B-connection range. As shown in Fig. 3.19(c), g's A-connection handles the control-bit part, and hence g has 3 middle vertices. The first is the "on-path" case when the control-bit is 0, so no interpretation of the controlled-bit is needed; the second is the "off-path" case; and the third represents the information "the control-bit was activated." Consequently, the B-connection groupings correspond to the respective three cases: the first has the Identity matrix; the second a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix); and the third is a CNOT structure of the form $CNOT(\frac{n}{2}, \blacksquare, j')$.

4. i lies in g's A-connection range, but j does not fall in g's range (Fig. 3.19(d)). This case is similar to Fig. 3.19(c), in that g's A-connection handles the control-bit part, and hence g has 3 middle vertices. The difference comes in the B-connections, which propagate the "states" of the middle vertices to g's 3 exit vertices (using the Identity matrix for both "on-path" and "the control-bit was activated" and a NoDistinctionProtoCFLOBDD for "off-path").

5. i lies in g's B-connection range, but j does not fall in g's range. As shown in Fig. 3.19(e), the bits in g's A-connection are not involved, and hence the A-connection is the Identity matrix. For the "on-path" middle vertex, g's B-connection is to a CNOT structure of the form $CNOT(\frac{n}{2}, i', \blacksquare)$. For the "off-path" middle vertex, g's B-connection is to a NoDistinctionProtoCFLOBDD.

6. j lies in g's A-connection range, but i does not fall in g's range (Fig. 3.19(f)). g's A-connection handles the part of the CNOT operation for flipping the

controlled-bit and has 2 exit vertices. The bits of g's B-connection are not involved; hence g's "off-path" middle vertex is a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix) and the "on-path" middle vertex is the Identity matrix.

7. j lies in g's B-connection range, but i does not fall in g's range (Fig. 3.20(a)). This case is similar to Fig. 3.19(f), except that the part of the CNOT operation for flipping the controlled-bit is displaced to the "on-path" B-connection ("on-path" from the Identity matrix in g's A-connection). The "off-path" B-connection is a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix).

8. There are two base cases of the recursive construction:

   - Calls of the form $CNOT(n, 1, \blacksquare)$ at level 1 (Fig. 3.20(b)). The grouping g for this case represents a sub-matrix similar to the identity matrix I_2, except that the second return edge of the second B-connection of g maps to a third (new) exit vertex of g (instead of mapping to the first exit vertex of g, as is the case for I_2). The third exit represents the information "the control-bit was activated" (i.e., the control-bit was 1), and hence, as indicated in Eqn. (3.33), for a matched-path to represent an element of the CNOT relation, there is an obligation to flip the value of the controlled-bit (elsewhere in the CFLOBDD).
   - Calls of the form $CNOT(n, \blacksquare, 1)$ at level 1 (Fig. 3.20(c)). Note from Fig. 3.19(c) that the initial construction of the form $CNOT(n, \blacksquare, j)$ (at some higher level) is attached to the third middle vertex of the parent grouping, which, in turn, has a return edge from the third exit of the parent grouping's A-connection $CNOT(n, i, \blacksquare)$. Thus, $CNOT(n, \blacksquare, j)$ only occurs in a path context in which it is known that the control-bit was activated.

     The $CNOT(n, \blacksquare, 1)$ grouping at level 1 interprets the controlled-bit. The proto-CFLOBDD used in this case is the same one found in both I_2 and X_2.

Inspection of Fig. 3.19(c), (f), and (g) reveals that such a proto-CFLOBDD is always connected to a level-2 grouping with return edges to middle or exit vertices that represent the "state-tuple" [off-path, on-path]. Consequently, each occurrence of a level-1 $CNOT(n, \blacksquare, 1)$ proto-CFLOBDD acts like the not matrix X_2.

Pseudo-code for the full algorithm is given in Appendix §I.

Complexity Every CFLOBDD that represents a CNOT relation consists of only a constant number of kinds of groupings: as shown in Figs. 3.19 and 3.20, the representation of CNOT uses at most two kinds of CNOT groupings at level 1 and seven kinds of CNOT groupings at levels ≥ 2, as well as proto-CFLOBDDs obtained from IdentityMatrixGrouping and NoDistinctionProtoCFLOBDD. Because at each level there are a constant number of groupings, each of constant size, the CFLOBDD representation of CNOT exhibits double-exponential compression compared to the size of the decision tree.[18]

A Special-Case Construction In some quantum algorithms, such as Simon's Algorithm, the bits are in two groups, x_i and y_i, $1 \leq i \leq n$, and the CNOT operation is performed for each pair (x_i, y_i). The same net result can be produced by creating (and then applying) a representation for the compound operation $\Pi_{i=1}^{n} CNOT(2n, i, n+i)$. This expression involves n matrix-multiplication operations. Unfortunately, if the bit ordering is $\langle x_1, \cdots, x_n, y_1, \cdots, y_n \rangle$, the size of the resulting CFLOBDD is exponential in n. In essence, the CFLOBDD's top-level A-connection has to "memorize" the exponential number of different possible combinations of x_i values so that the information can be correlated with the y_i values in the B-connection.

[18] The general case—$CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$, where j is allowed to be less than i—would still have at each level a constant number of kinds of groupings, each of constant size. Consequently, the general case also exhibits double-exponential compression compared to the size of the decision tree.

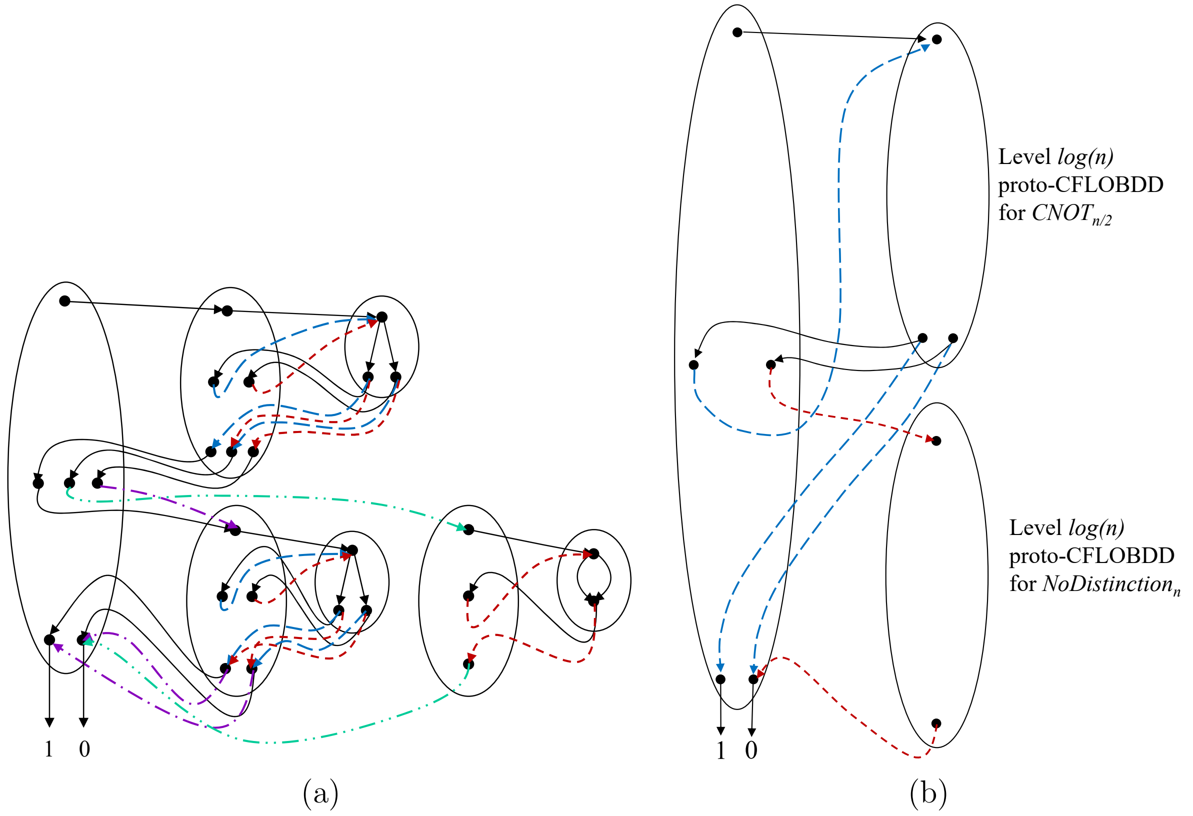


Figure 3.21: CFLOBDD representation of $CNOT_n = CNOT_2^{\otimes n}$ with variable ordering $\langle x_1, y_1, x_2, y_2, \cdots, x_n, y_n \rangle$. (a) Base case (level 2): $CNOT_2$ for 2 bits, with the first bit as the control-bit and second bit as the controlled-bit. (b) Grouping structure used for levels greater than 2.

Fortunately, changing to the interleaved-variable ordering $\langle x_1, y_1, x_2, y_2, \cdots, x_n, y_n \rangle$ leads to an efficient CFLOBDD representation of $\Pi_{i=1}^{n}(CNOT(2n, i, n+i)$. With the interleaved-variable ordering, after reassigning the index numbers $[1 \ldots 2n]$ to the variables in the new order, $CNOT_n$ changes to the simpler expression

$$CNOT_n = \Pi_{\substack{i=1 \\ i+=2}}^{2n} CNOT(2n, i, i+1). \tag{3.34}$$

In Eqn. (3.34), each CNOT operation is between *adjacent* bits, and hence Eqn. (3.34) can be rewritten as

$$CNOT_n = \bigotimes_{i=1}^{n} CNOT_2 = CNOT_2^{\otimes n} \tag{3.35}$$

Moreover, we do not even have to perform the n explicit Kronecker-product operations of Eqn. (3.35) (or even $\log n$ Kronecker products) because $CNOT_n$ can be constructed directly (cf. item 1b in §2.3.1). Fig. 3.21 depicts the CFLOBDD representation of $CNOT_n = CNOT_2^{\otimes n}$, which has $\log n + 1$ levels and $2n$ Boolean variables, and the pseudo-code for the construction algorithm can be found in §I.

3.8.1.6 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a linear transformation that is somewhat similar to the discrete Fourier transform. In matrix form, it looks like[19]

$$QFT_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where $\omega = e^{2\pi i/N}$.

We use the algorithm presented in Nielsen and Chuang (2000) for the application of QFT to a state vector. More formally, we use the following steps to apply QFT for n qubit vector:

1. Start with a random vector e_x. (We use a randomly chosen basis vector as input in our experiments).

2. Apply the Hadamard matrix H_2 to the ith qubit, where $i \to \{n \dots 1\}$.

3. For every i, apply a Controlled-Phase Gate CP from qubit $j \to \{1..i-1\}$ to i with a phase $= \frac{\pi}{2^{i-j}}$.

[19] In the description of QFT, we vary from the subscript convention explained at the beginning of §2.3. Here, a gate matrix acting on n qubits (which is of size $2^n \times 2^n$) is denoted using a subscript N, where N denotes 2^n.

4. Finally, apply $n/2$ Swap Gates between i and $n - i$, where $i \to \{1..n/2\}$.

The construction of Controlled-Phase Gate and Swap Gate is given in Appendix §J.

3.8.2 Quantum Algorithms

In this section, we briefly describe the quantum algorithms that are used in the experiments in §3.9.2.2. The ingredients of the algorithms are qubits, quantum gates, and measurements of qubits. Each of the states or operations can be viewed as algebraic operations on vectors and matrices. The mapping between the various aspects of the quantum algorithms and their corresponding algebraic operations is as follows:

- Sequence of n qubits: Unit vector of dimension $2^n \times 1$ (called a state vector)
- Quantum gate: Unitary matrix
- Augmenting a sequence of qubits with additional qubits: Kronecker product of vectors
- Application of a quantum gate to a sequence of qubits: Matrix-vector multiplication
- Measurement of qubits: Sampling from a distribution obtained from the vector

We now discuss quantum algorithms as algebraic operations.

3.8.2.1 GHZ Algorithm

The Greenberger–Horne–Zeilinger (GHZ) state is the following ("entangled") state vector for 3 qubits (i.e., a unit vector of size 8×1):

$$GHZ_3 = \frac{e_{000} + e_{111}}{\sqrt{2}}$$

We extend the concept to n qubits by defining[20]

$$GHZ_n = \frac{e_{0^n} + e_{1^n}}{\sqrt{2}}$$

We have used the algorithm given by Yu and Palsberg Yu and Palsberg (2021) for obtaining the GHZ state for n bits, which is as follows:

1. Prepare an initial state $e_{0^{n+1}}$ (the standard-basis vector of $n+1$ bits with a 1 in the first position and 0s elsewhere).
2. Apply the Hadamard matrix H_{2n} on the first n bits and the Not matrix X_2 on the $(n+1)^{st}$ bit.
3. Apply n CNOTs of the form $CNOT(n+1, i, n+1)$, where $i \to \{1..n\}$.
4. Apply the Hadamard matrix $H_{2(n+1)}$ $(= H_{2n} \otimes H_2)$ on all $n+1$ bits.
5. Measure the first n bits to obtain a string of n 0s or n 1s.

Steps (1)–(4) can be expressed in matrix-vector notation as follows:

$$(H_{2n} \otimes H_2)(\Pi_{i=1}^{n} CNOT(n+1, i, n+1))(H_{2n} \otimes X_2)(e_{0^{n+1}}) \tag{3.36}$$

Eqn. (3.36) is expressed in a way that uses vectors indexed by $n+1$ Boolean variables, and matrices indexed by $2n+2$ Boolean variables. Whereas a BDD implementation can directly encode Eqn. (3.36), CFLOBDDs are a representation of functions in which the number of Boolean variables must be a power of 2. For this reason, in a CFLOBDD implementation of *GHZ*, vectors and matrices are augmented so that vectors have $n-1$ dummy index variables and matrices have $2n-2$ dummy index variables. Taking into account these dummy variables, what is actually computed is the following:

$$(H_{2n} \otimes H_2 \otimes I^{\otimes n-1})(\Pi_{i=1}^{n} CNOT(n+1, i, n+1))(H_{2n} \otimes X_2 \otimes I^{\otimes n-1})(e_{0^{2n}}) \tag{3.37}$$

[20]Recall that we use e_{0^n} and e_{1^n} denote the standard-basis vectors $e_{\underbrace{0\ldots0}_{n\ copies}}$ and $e_{\underbrace{1\ldots1}_{n\ copies}}$, respectively.

By properties of Kronecker product,

$$(H_{2n} \otimes X_2 \otimes I^{\otimes n-1})(e_{0^{2n}}) = (H_{2n} \times e_{0^n}) \otimes (X_2 \times e_0) \otimes (I^{\otimes n-1} \times e_{0^{n-1}}).$$

Because the matrix-vector product $H_{2^n} \times e_{0^n}$ results in a vector consisting of all ones, we can avoid performing an explicit multiplication and instead directly create a CFLOBDD whose top-level grouping is a `NoDistinctionProtoCFLOBDD` and whose terminal value is $\frac{1}{\sqrt{2^n}}$. In the implementation of these algorithms, a matrix-vector multiplication is implemented by first converting the vector $2^k \times 1$ to a matrix of size $2^k \times 2^k$ by padding with zeros (§3.6.6), and then performing matrix multiplication (§3.6.7).

In step (5), the terminal values (which represent amplitudes) are squared, and then the bit-strings of indices are sampled from the CFLOBDD based on the values of the squared amplitudes (which represent probabilities), as explained in §3.6.8.

3.8.2.2 Bernstein-Vazirani Algorithm

The problem that the Bernstein-Vazirani (BV) algorithm solves is as follows:

> Given an oracle that implements a function $f : \{0,1\}^n \rightarrow \{0,1\}$ in which $f(x)$ is promised to be the dot product, mod 2, between x and a secret string $s \in \{0,1\}^n$—i.e., $f(x) = x_1 \cdot s_1 \oplus x_2 \cdot s_2 \oplus \cdots \oplus x_n \cdot s_n$—find s.

Given an oracle $U_{2(n+1)}$ that implements f, such that $U(xy) = x(f(x) \oplus y)$, the BV algorithm is as follows:

1. Prepare an initial state $e_{0^{n+1}}$ (the standard-basis vector of $n+1$ bits with a 1 in the first position and 0s elsewhere).
2. Apply the Hadamard matrix H_{2n} on the first n bits.
3. Apply the oracle $U_{2(n+1)}$ to the current state.
4. Finally, apply the Hadamard matrix H_{2n} on the first n bits.

5. Measure the first n bits to obtain the string s.

Steps (1)–(4) of the algorithm can be expressed as

$$(H_{2n} \otimes I_2)(U_{2(n+1)})(H_{2n} \otimes I_2)(e_{0^{n+1}}) \quad (3.38)$$

As in §3.8.2.1, to implement Eqn. (3.38) with CFLOBDDs, we introduce dummy index variables so that the total number of index variables is a power of 2. Also, in the term $(H_{2n} \otimes I_2)(e_{0^{n+1}}) = (H_{2n} \times e_{0^n}) \otimes (I_2 \times e_0)$, the multiplication $(H_{2n} \times e_{0^n})$ can be avoided by directly creating a CFLOBDD whose top-level grouping is a `NoDistinctionProtoCFLOBDD` and whose terminal value is $\frac{1}{\sqrt{2^n}}$.

3.8.2.3 Deutsch–Jozsa algorithm

The algorithm solves the following problem:

> Given an oracle that implements a function $f : \{0, 1\}^n \rightarrow \{0, 1\}$, where f is promised to be either a constant function (0 on all inputs or 1 on all inputs) or a balanced function (returns 1 for half of the input domain and 0 for the other half), determine if f is balanced or constant.

The oracle takes the form of a matrix $U_{2(n+1)}$, for which $U(xy) = x(f(x) \oplus y)$. The steps of the Deutsch–Jozsa (DJ) algorithm are as follows:

1. Prepare an initial state e_{0^n1} of $n + 1$ bits (the standard-basis vector of $n + 1$ bits with a 1 in the second position and 0s elsewhere).

2. Apply the Hadamard matrix $H_{2n} \otimes H_2$ on first $n + 1$ bits.

3. Apply the oracle $U_{2(n+1)}$ to the current state.

4. Finally, apply the Hadamard matrix H_{2n} to the first n bits.

5. Measure the first n bits to obtain the string s. If $s = e_{0^n}$, then f is constant; otherwise, f is balanced.

Steps (1)–(4) can be expressed as:

$$(H_{2n} \otimes I_2)(U_{2(n+1)})(H_{2n} \otimes H_2)(e_{0^n} \otimes e_1).$$

3.8.2.4 Simon's Algorithm

The problem that Simon's algorithm addresses is as follows:

> One is given a function $f : \{0,1\}^n \to \{0,1\}^n$, where f is promised to satisfy the property that there is a "hidden vector" $s \in \{0,1\}^n$ such that, for all x and y, $f(x) = f(y)$ if and only if $x = y \oplus s$. The goal is to find the hidden vector s.

Given an oracle U_{2n} such that $U(x) = f(x)$, satisfying the property $\exists s \in \{0,1\}^n$ such that $\forall x, y\ U(x) = U(y)$ if and only if $x = y \oplus s$, the algorithm for finding s is as follows:

1. Initialize the set of equations E to the empty set
2. Prepare an initial state $e_{0^{2n}}$ of $2n$ bits (the standard-basis vector of $2n$ bits with a 1 in the first position and 0s elsewhere).
3. Apply the Hadamard matrix H_{2n} on the first n bits.
4. Apply the oracle U_{2n} on the first n bits.
5. Successively apply the matrices $CNOT(2n, i, n+i)$, for $i \to \{1..n\}$.
6. Apply the oracle U^*_{2n}, which is the conjugate of U_{2n}, on the first n bits.
7. Apply the Hadamard matrix H_{2n} on the first n bits.
8. Measure the first n bits to obtain x.
9. Add the equation $x \cdot s = 0$ to equation set E.

10. Repeat steps (2)–(9) to obtain $O(n)$ equations.

11. With high probability, the solution to the set of equations E is s.

Steps (2)–(10) operate on quantum states of $2n$ qubits. Call the first n bits the x bits, and the second n bits the y bits. Conventionally, one considers the bits to be ordered as follows: $\langle x_1, \cdots, x_n, y_1, \cdots, y_n \rangle$.

Steps (2)–(7) can be written as follows:

$$(H_{2n} \otimes I_{2n})(U_{2n}^* \otimes I_{2n})(\Pi_{i=1}^n(CNOT(2n, i, n+i)))((U_{2n} \otimes I_{2n}))(H_{2n} \otimes I_{2n})(e_{0^{2n}})$$

Using Eqn. (3.34), and the interleaved Kronecker product ($\otimes_i$) discussed in §3.6.5.2 (Alg. 18), we can rewrite the above expression as follows:

$$(H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \tag{3.39}$$

Note that in Eqn. (3.39), we have changed to the bit ordering to $\langle x_1, y_1, x_2, y_2, \cdots, x_n, y_n \rangle$.

Now consider the second, third, and fourth multiplicands in Eqn. (3.39):

$$(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n}).$$

Reading these terms right to left,

- first, the oracle U_{2n} is applied on the x bits, which yields $f(x)$;

- second, $CNOT_n$ copies the value in the x bits to the y bits, leading to the y bits also representing $f(x)$;

- third, U_{2n}^* is applied to the x bits, to turn the values in the x bits from $f(x)$ back to the original value of x.

However, we can achieve the same result by first copying the value of the x bits to the y bits by applying $CNOT_n$, and then applying U_{2n} directly on the y bits, which can be expressed as $(I_{2n} \otimes_i U_{2n})(CNOT_n)$. Thus, Eqn. (3.39) can be re-written as

$$(H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \tag{3.40}$$

Formally, to show that Eqns. (3.39) and (3.40) are equal, we use two properties of Kronecker product:

1. $A \otimes (B + C) = (A \otimes B) + (A \otimes C)$
2. $(A \otimes B)(C \otimes D) = (AC \otimes BD)$

Starting from Eqn. (3.39), we have

$$\begin{aligned}
&(H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) && (3.39)\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^n} \otimes_i e_{0^n})\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n}e_{0^n} \otimes_i I_{2n}e_{0^n})\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(\Sigma_x(e_x \otimes_i e_{0^n}))\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(\Sigma_x(U_{2n}e_x \otimes_i I_{2n}e_{0^n}))\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(CNOT_n)(\Sigma_x(e_{f(x)} \otimes_i e_{0^n}))\\
&= (H_{2n} \otimes_i I_{2n})(U^*_{2n} \otimes_i I_{2n})(\Sigma_x(e_{f(x)} \otimes_i e_{f(x)}))\\
&= (H_{2n} \otimes_i I_{2n})(\Sigma_x(U^*_{2n}e_{f(x)} \otimes_i I_{2n}e_{f(x)}))\\
&= (H_{2n} \otimes_i I_{2n})(\Sigma_x(e_x \otimes_i e_{f(x)})) && (3.41)
\end{aligned}$$

Starting from Eqn. (3.40), we have

$$
\begin{aligned}
& (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) && (3.40)\\
= & (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^n} \otimes_i e_{0^n})\\
= & (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} e_{0^n} \otimes_i I_{2n} e_{0^n})\\
= & (H_{2n} \otimes_i U_{2n})(CNOT_n)(\Sigma_x (e_x \otimes_i e_{0^n}))\\
= & (H_{2n} \otimes_i U_{2n})(\Sigma_x (e_x \otimes_i e_x))\\
= & (H_{2n} \otimes_i I_{2n})(I_{2n} \otimes_i U_{2n})(\Sigma_x (e_x \otimes_i e_x))\\
= & (H_{2n} \otimes_i I_{2n})(\Sigma_x (I_{2n} e_x \otimes_i U_{2n} e_x))\\
= & (H_{2n} \otimes_i I_{2n})(\Sigma_x (e_x \otimes_i e_{f(x)})) && (3.42)
\end{aligned}
$$

Eqns. (3.41) and (3.42) are identical, and hence Eqn. (3.40) can be used in place of Eqn. (3.39).

For step (11), our implementation uses the algorithm presented in (Elder et al., 2014, §2.1) for solving the set E of Boolean linear equations.

The implementation also illustrates two of the advantages of simulation discussed in §2.3.1:

- The discussion above about choosing to interleave the bits of x and y, and replacing $\Pi_{i=1}^{n}(CNOT(2n, i, n+i))$ with $CNOT_n$ is in the spirit of item 1b.
- In accordance with item 1c, the implementation does not have to perform steps (2)–(7) for each sampling step. Instead, it performs steps (2)–(7) *once* to build up an appropriate CFLOBDD, and then samples the desired number of times from that structure.

3.8.2.5 Shor's Algorithm

Shor's Algorithm aims to find prime factors of a given integer N. We use the following algorithm:

1. Pick a random number $1 < a < N$.
2. Compute $K = \gcd(a, N)$. If $K \neq 1$, then K is a nontrivial factor of N, so we are done.
3. Otherwise, use the quantum period-finding algorithm (see below) to find r, which denotes the period of the function $f(x) = a^x \bmod N$. Equivalently, r is the smallest positive integer that satisfies $a^r \equiv 1 \bmod N$.
4. If r is odd, then go back to step (1).
5. If $a^{r/2} = -1 \bmod N$, then go back to step (1).
6. Otherwise, both $\gcd(a^{r/2} + 1, N)$ and $\gcd(a^{r/2} - 1, N)$ are nontrivial factors of N, so we are done.

The quantum period-finding algorithm is as follows:

1. Start with $3n$ qubits with the first $2n$ qubits in state $e_{0^{2n}}$ and the last n qubits in state $e_{0^{n-1}1}$.
2. For each qubit $i \to \{N..1\}$, apply the controlled gate $U_a^{2^{(N-i-1)}}$ on the last n qubits, where $U_a(x) = ax \bmod N$.
3. Apply Inverse QFT on the first $2n$ qubits.
4. Measure the first $2n$ qubits to find r as discussed in (Lipton and Regan, 2014, §11.6).

3.8.2.6 Grover's Algorithm

The algorithm solves the problem of finding a needle in a haystack. More formally,

> Given a function $f : \{0, 1, \cdots, N-1\} \to \{0, 1\}$, the algorithm finds x such that $f(x) = 1$.

Here, we assume that there is only one x such that $f(x) = 1$. The algorithm uses an oracle U_{2n}, which implements f for the "needle" w as

$$U_w(x) = \begin{cases} x & \text{if } x = w \\ -x & \text{otherwise} \end{cases}$$

Given this oracle, the algorithm for finding s is as follows:

1. Prepare an initial state e_{0^n} (the standard-basis vector of n bits with a 1 in the first position and 0s elsewhere).

2. Apply the Hadamard matrix H_{2n} on the initial state.

3. Apply the oracle U_w to the current state.

4. Apply the Grover diffusion operator $U_s = H_{2n}(2e_{0^n} \otimes_{outer} e_{0^n} - I_{2^n})H_{2n}$, where $\otimes_{outer}$ denotes the outer product of two vectors.

5. Repeat steps (3)–(4) $\lceil \frac{\pi}{4}\sqrt{N} \rceil$ times.

6. Measure the n qubits to obtain the string s.

Steps (1)–(5) can be written concisely as follows:

$$\left(\Pi_{i=1}^{\lceil \frac{\pi}{4}\sqrt{N} \rceil} U_s U_w\right) (H_{2n}(e_{0^n}))$$

As mentioned in §3.8.2.6, one of the advantages of simulation is that we can re-associate the matrix multiplications in steps (1)–(4) to compute $\left(\Pi_{i=1}^{\lceil \frac{\pi}{4}\sqrt{N} \rceil} U_s U_w\right)$ as an explicit quantity, which can be done using repeated squaring instead of as a sequence of multiplications. This approach provides a more efficient approach to steps (3)–(4). We can also use optimizations like those discussed earlier, e.g., performing (1)–(2) by means of a direct construction of the desired vector of all-1s. We can also create a representation of the diffusion operator U_s directly, rather than performing the computation given in step (4). In particular, the construction is almost

the same as the identity-matrix construction in Alg. 27: one would use subroutine `IdentityMatrixGrouping`, but at top level the value tuple would be $[\frac{2}{N} - 1, \frac{2}{N}]$ (instead of $[1, 0]$).

3.9 Evaluation

In this section, we explain our experimental setup and describe the experiments we carried out, which were designed to address the following research questions:

RQ1: Do theoretical guarantees of *double-exponential compression* by CFLOBDDs allow them to represent substantially larger Boolean functions than BDDs?

RQ2: Do CFLOBDDs outperform BDDs when used for quantum simulation (in terms of time and space)?

RQ3: Do CFLOBDDs outperform BDDs when used in hardware verification benchmarks (in terms of time and space)?

3.9.1 Experimental Setup

We compared our implementation of CFLOBDDs[21] against a widely used BDD package, CUDD Somenzi (2012) (version 3.0.0),using CUDD's C++ interface. The metrics are (i) execution time, and (ii) space (node counts in the case of BDDs; vertex counts + edge counts in the case of CFLOBDDs). We ran all experiments on AWS machines: t2.xlarge machines with 4 vCPUs, 16GB of memory, and a stack size of 8192KB, running on Ubuntu OS.

For RQ1 (§3.9.2.1), we used a collection of synthetic benchmarks, and compared the performance of CFLOBDDs against (i) CUDD with a static variable ordering (similar to the one used in the CFLOBDDs), (ii) CUDD with dynamic variable reordering, and (iii) Sentential Decision Diagrams (SDDs) Darwiche (2011) (which can

[21] The implementation is available at `https://github.com/trishullab/cflobdd`.

also be exponentially more succinct than BDDs), using Python package PySDD Meert and Choi (2018) (version 0.1).

For RQ2 (§3.9.2.2), we used a set of quantum-simulation benchmarks, and again compared the performance of CFLOBDDs against CUDD (version 3.0.0). For the quantum benchmarks, we did not enable dynamic variable reordering for BDDs because we could not retrieve the correct order of the output bits for a sampled string.

Five of the quantum benchmarks—BV, DJ, Simon's algorithm, Shor's algorithm, and Grover's algorithm—use oracles for that either directly or indirectly incorporate the answer sought. Our methodology is standard for quantum-simulation experiments. Each benchmark uses a pre-processing step to create the CFLOBDD/BDD that represents the oracle. In each run, an answer is first generated randomly, and then the CFLOBDD/BDD that represents the oracle is constructed. Knowledge about the answer is used only during oracle construction. Thereafter, the quantum algorithm proper is simulated; these steps have no access to the pre-chosen answer (other than the ability to perform operations on the oracle, treated as a unitary matrix). The final step of running the benchmark is to check that the quantum algorithm obtained the correct answer.

We could not run the quantum benchmarks with SDDs because SDDs do not support multi-terminal values. However, we ran the quantum benchmarks using Quimb Gray (2018), a quantum-simulation library that uses tensor networks.

Extensions to CUDD For the RQ2 experiments, we had to extend CUDD in two ways to be able to simulate quantum circuits using CUDD data structures:

1. CUDD supports *algebraic decision diagrams* (ADDs), which are multi-terminal BDDs with a value from a semiring at each terminal node. We had to extend CUDD with a semiring that was not part of the standard CUDD distribution (§3.9.1.1). For the corresponding experiments with CFLOBDDs, we used the same semiring for the terminal values of CFLOBDDs.

2. To allow quantum measurements to be carried out, we extended ADDs to support path sampling (i.e., selection of a path, where the probability of returning a given path is proportional to a function of the path's terminal value).

3.9.1.1 Algebraic Decision Diagrams with Complex-Number Leaves

To use CUDD on most of the quantum benchmarks, we modified the ADD datatype and related functions to use multi-precision-floating-point complex numbers Fousse et al. (2007).

For the "Quantum Fourier Transform" and "Shor's Algorithm" benchmarks, one only needs to represent a known set of complex roots of unity, so we used a different custom ADD datatype, *dtype*, defined as follows:

```
typedef struct dtype {
    int val;
    int size;
    mpfr_t real;
    mpfr_t imag;
} dtype;
typedef dtype CUDD_VALUE_TYPE;
```

Here, "size" represents the number of roots of unity, and a value i for "val" represents the ith root of unity with respect to the current "size." For example, if size = 4, then val $\in \{0, 1, 2, 3\}$. Multiplying two *dtype*s—t_1 and t_2 with size = 4, where val of t_1 = 2 and val of t_2 = 3—produces *dtype* t_3 with size = 4 and val = (2 + 3) % 4 = 1. This representation encodes $\omega^2 \times \omega^3 = \omega$, where $\omega^k = e^{\frac{2k\pi}{4}}$ is a primitive 4th root of unity. We used this representation to compute the entire quantum Fourier transform, and then filled in the values for real and imag at the last step, where real = $\cos(\frac{2\pi * val}{size})$ and imag = $\sin(\frac{2\pi * val}{size})$.

3.9.1.2 Sampling in BDDs.

An important step in quantum simulation is measurement of the output bits, which is equivalent to sampling a path from the BDD structure according to a probability distribution obtained from the terminal values. In particular, the terminal values represent so-called "amplitudes," and the corresponding probability distribution is based on the squares of the amplitudes.

Path Counting Suppose that the BDD has k terminals. To sample a path based on the squares of the amplitudes, we need to know, for each node n in the BDD, and each terminal position t_i, $1 \leq i \leq k$, the number of paths that lead from n to t_i. This information can be computed by generalizing the method of Ball and Larus for counting the number of paths in a DAG Ball and Larus (1996) from the case of a single exit-node to k exit-nodes (while also accounting for ply-skipping in a BDD):

- Each "*path-count*" is a k-dimensional vector c. (For convenience, we index the components of c as $c_1, \ldots, c_k$.)
- Path-counts are computed bottom-up, starting from the terminal positions. The path-count at terminal position i has a 1 at index-position i and zeros elsewhere, signifying that (i) there is exactly 1 path from terminal node i to itself, and (ii) there are no paths from terminal node i to any of the other terminal nodes.
- When no plies are skipped in going from a node n to each of its two children, the path-count at n is the vector addition of the left-child and right-child path-counts.
- If $\Delta \mathit{left}$ plies are skipped in going from n to the left child (with path-count c_l), and $\Delta \mathit{right}$ plies are skipped in going to the right child (with path-count c_r), the path-count at n is $2^{\Delta \mathit{left}} c_l + 2^{\Delta \mathit{right}} c_r$.
- If Δ plies have been skipped at the apex of the DAG (with path-count c), the BDD's overall path-count is $2^{\Delta} c$.

Sampling Once path-counts are in hand, we sample a path as follows:

- Let $n.pc_i$ denote the path-count at a node n. Let $n.lchild$ and $n.rchild$ denote the two children of n, and let $n.\Delta left$ and $n.\Delta right$ denote the number of plies skipped in going to the left child and right child of n, respectively.

- For convenience, if Δ plies have been skipped at the apex a of the DAG, we assume that a new root node r is added whose left-child and right-child both point to a. $r.pc$ is set to $2^{\Delta} a.pc$, and both $r.\Delta left$ and $r.\Delta right$ are set to $\Delta - 1$. Otherwise, the root r is a.

- Let $p = \langle p_1, \ldots, p_k \rangle$, denote the vector of squared amplitudes at the k terminal positions. The probability of choosing a path from r to terminal position i is $p_i * r.pc_i$.

- Randomly choose a value i based on the probabilities $\langle p_1 * r.pc_1, \ldots, p_k * r.pc_k \rangle$. Henceforth, we only consult the ith components of path-counts of the BDD's nodes.

- We work down the BDD from r to terminal position i, accumulating a path-string in π, which is initially set to ϵ. Starting at r, and then at each subsequently visited node n until terminal position i is reached, take the following steps:

  - If both $n.lchild.pc_i \neq 0$ and $n.rchild.pc_i \neq 0$, randomly choose one of the children in relative proportion to $2^{n.\Delta left} n.lchild.pc_i$ and $2^{n.\Delta right} n.rchild.pc_i$. If the left child is chosen, append "0" followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta left}$ to π; otherwise, append "1" followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta right}$ to π.

  - If $n.lchild.pc_i = 0$, append "1" followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta right}$ to π.

– If $n.rchild.pc_i = 0$, append "0" followed by a string chosen uniformly from $\{0,1\}^{n.\Delta left}$ to π.

Set n to the child chosen above.

3.9.2 Benchmarks and Experimental Results

3.9.2.1 RQ1: Do theoretical guarantees of *double-exponential compression* by CFLOBDDs allow them to represent substantially larger Boolean functions than BDDs?

We used the following three benchmarks to compare the execution time and memory usage (as vertex count + edge count) of CFLOBDDs against BDDs and SDDs.

- $XOR_n = \bigoplus_{i=1}^{n} x_i$
- $MatMult_n = (H_n I_n + X_n H_n + I_n X_n)$, where H_n is the Hadamard matrix, I_n is the Identity matrix, and X_n is the NOT matrix of size 2^{n-1} x 2^{n-1}. (The aim of the benchmark is to test the performance of the matrix-multiplication and addition operations.)
- $ADD_n(X,Y,Z) \stackrel{\text{def}}{=} Z = (X + Y \mod 2^{n/4})$, where X, Y, and Z are $n/4$-bit integers.

Tab. 3.2 shows the performance of CFLOBDDs, BDDs (with and without dynamic reordering enabled), and SDDs within the 15-minute timeout threshold. For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1GB. For the *ADD* benchmark, BDDs (both with and without dynamic reordering) and SDDs ran out of memory within the 15-minute timeout threshold for problems with sufficiently many variables, even with such a large stack. (BDDs with dynamic reordering produced out-of-memory errors for #variables $\geq 2^{24}$: the first step in the computation is to allocate the variables, which by itself leads to memory exhaustion

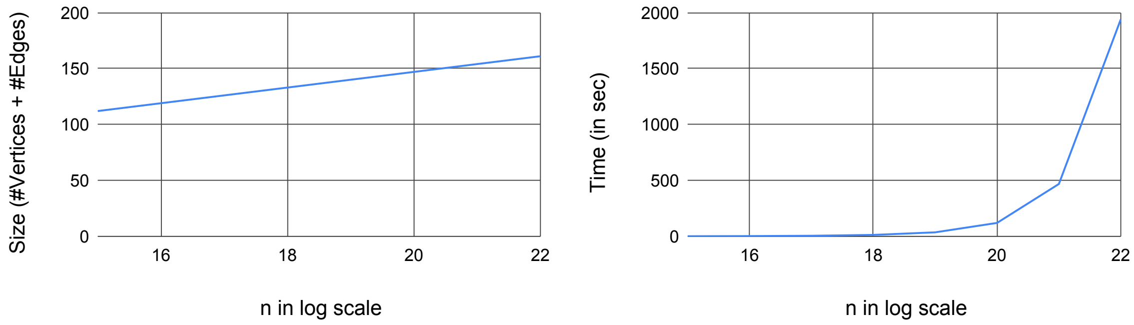
Benchmark	#Boolean Variables (n)	CFLOBDD				BDD		BDD (reorder)		SDD	
		#Vertices	#Edges	Total	Time (sec)	#Nodes	Time (sec)	#Nodes	Time (sec)	#Nodes	Time (sec)
XOR_n	2^{15}	16	96	112	**0.99**	32769	551.27	32769	587.11	131066	3.91
	2^{16}	17	102	119	**2.18**	Timeout (15min)				262138	8.57
	2^{17}	18	108	126	**5**					524282	18.71
	2^{18}	19	114	133	**12.75**					1048570	38.63
	2^{19}	20	120	140	**36.06**					2097146	82.03
	2^{20}	21	126	147	**122.97**					4194298	191.68
$MatMult_n$	2^{15}	84	1053	1137	**0.002**	294890	57.33	294890	156.42	Not Applicable	
	2^{16}	90	1125	1137	**0.004**	589802	186.27	593122	446.19		
	2^{17}	96	1197	1293	**0.007**	1179626	739.66	Timeout (15min)			
	2^{18}	102	1269	1371	**0.017**	Timeout (15min)					
	2^{19}	108	1341	1449	**0.043**						
	2^{20}	114	1413	1527	**0.118**						
	2^{21}	120	1485	1605	**0.343**						
	2^{22}	126	1557	1683	**1.238**						
	2^{23}	132	1629	1761	**4.936**						
	2^{24}	138	1701	1839	**19.37**						
	2^{25}	144	1773	1917	**78.98**						
	2^{26}	150	1845	1995	**317.27**						
	2^{27}	Timeout (15min)									
ADD_n	2^{17}	80	574	654	**<0.001**	131073	0.035	132405	80.24	393152	7.72
	2^{18}	85	610	695	**0.001**	262145	0.065	263477	280.79	786364	13.82
	2^{19}	90	646	736	**0.001**	524289	0.148	Timeout (15min)		1572792	29.72
	2^{20}	95	682	777	**0.001**	1048577	0.293			3145652	66.26
	2^{21}	100	718	818	**0.001**	2097153	1.368			6291376	138.40
	2^{22}	105	754	859	**0.001**	4194305	1.155			12582828	359.26
	2^{23}	110	790	900	**0.002**	8388609	3.316			Out of Memory	
	2^{24}	115	826	941	**0.003**	Out of Memory					
	2^{25}	120	862	982	**0.003**						
	⋮	⋮	⋮	⋮	⋮	Out of Memory					
	$2^{2^{21}}$	10485755	75497434	85983189	**113.99**						
	$2^{2^{22}}$	20971515	150994906	171966421	**385.75**						
	$2^{2^{23}}$	Timeout (15min)									

Table 3.2: Performance of CFLOBDDs against BDDs, BDDs with dynamic reordering, and SDDs on the synthetic benchmarks for different numbers of Boolean variables. (For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1GB.)

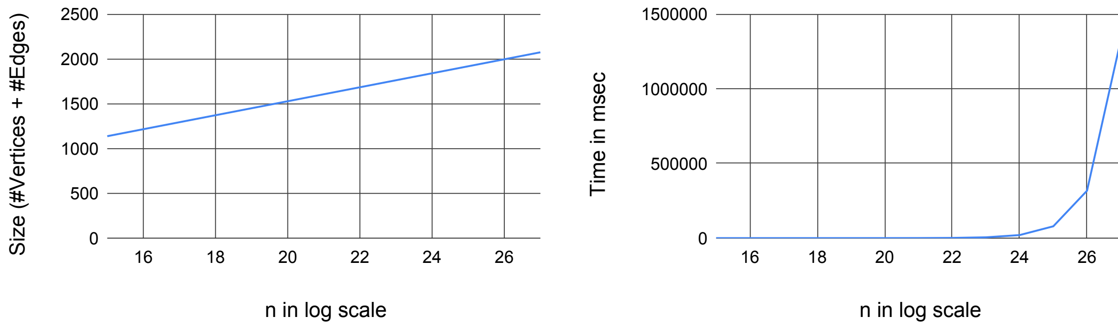
for 2^{24} variables and beyond.) Note that, for SDDs, benchmark $MatMult_n$ is not applicable because SDDs do not handle non-Boolean values.

Because of a slightly technical alignment issue, our CFLOBDD representations of ADD_n deliberately waste one-quarter of the Boolean variables (as *dummy variables*). To make a fair comparison, our BDD and SDD encodings of ADD_n use only three-quarters of the Boolean variables indicated in column two of Tab. 3.2.

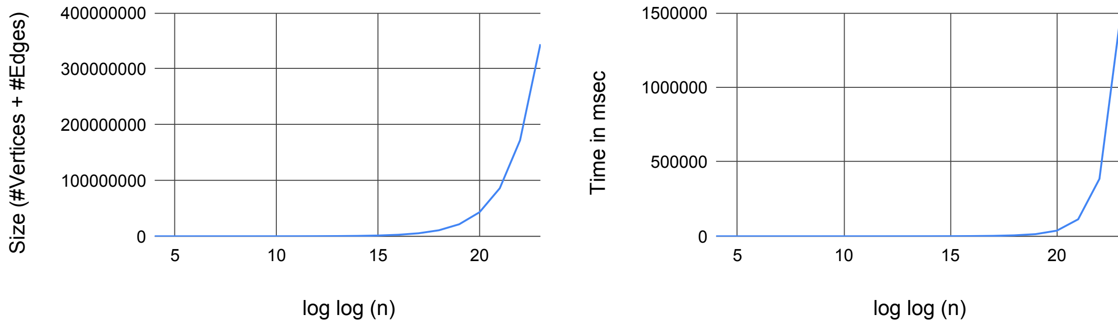
To understand how large a Boolean function could be created using CFL-



(a) XOR_n: Size and time vs. n (on a log scale)



(b) $MatMult_n$: Size and time vs. n (on a log scale)



(c) ADD_n: Size and time vs. n (on a $\log\log$ scale)

Figure 3.22: CFLOBDD performance with a timeout of ninety minutes. Note that in (c) the number of Boolean variables is on a $\log\log$ scale.

OBDDs (as a function of the number of Boolean variables),[22] we also measured the performance of the CFLOBDD implementation on the micro-benchmarks using a timeout of ninety minutes.

Fig. 3.22 shows graphs of size (#vertices + #edges) and time versus the number of Boolean variables for the three benchmarks.[23] Fig. 3.22a shows the graphs for XOR_n. In these graphs, time is in seconds, and the number of Boolean variables is on a log scale. We were able to construct XOR_n with up to $2^{22} = 4{,}194{,}304$ variables. Fig. 3.22b and Fig. 3.22c show the graphs for $MatMult_n$ and ADD_n, respectively. In these graphs, time is in milliseconds, and the number of Boolean variables is on a log scale for $MatMult_n$ and a $\log \log$ scale for ADD_n. We were able to construct $MatMult_n$ with up to $2^{27} = 134{,}217{,}728$ variables and ADD_n with up to $2^{2^{23}} = 2^{8{,}388{,}608}$ $\cong 4.27 \times 10^{2{,}525{,}222}$ variables, which comes to $0.75 \times 2^{8{,}388{,}608} \cong 3.12 \times 10^{2{,}525{,}222}$ after removing dummy variables.

Findings. CFLOBDDs performed better than BDDs and SDDs, both in terms of time and memory. For the benchmarks with more than 2^{18} Boolean variables, BDDs had memory issues. Using CFLOBDDs, it was also possible to construct representations of the benchmark functions with astounding numbers of Boolean variables: $2^{22} = 4{,}194{,}304$ for XOR_n; $2^{27} = 134{,}217{,}728$ for $MatMult_n$; and $0.75 \times 2^{8{,}388{,}608} \cong 3.12 \times 10^{2{,}525{,}222}$ for ADD_n. These results support the claim that CFLOBDDs can provide substantially better compression of Boolean functions than BDDs.

3.9.2.2 RQ2: Do CFLOBDDs outperform BDDs when used for quantum simulation (in terms of time and space)?

Tabs. 3.3 and 3.4 show the performance of CFLOBDDs and BDDs when simulating several well-known quantum algorithms, which are discussed in §3.8. In each case, for both CFLOBDDs and BDDs, we used the interleaved-variable ordering.

[22] The stack size was increased to 1GB for the runs with more than $2^{2^{15}}$ Boolean variables.

[23] Tab. 3.2 shows the comparison of CFLOBDDs, BDDs, and SDDs for examples with a 15-minute timeout. In contrast, Fig. 3.22 shows the results of the stress test that we performed, where we gave the CFLOBDD implementation a 90-minute timeout.

Benchmark	#Qubits	#Boolean Variables	CFLOBDD				BDD	
			#Vertices	#Edges	Total	Time (sec)	#Nodes	Time (sec)
GHZ	16	32	35	207	242	0.005	**36**	**0.003**
	32	64	43	255	298	**0.007**	**68**	0.008
	64	128	51	303	354	**0.010**	**131**	0.031
	128	256	59	351	410	**0.015**	**259**	0.143
	256	512	67	399	**466**	**0.027**	515	4.9
	512	1024	75	447	**522**	**0.051**	1028	44
	1024	2048	83	495	**578**	**0.107**	Timeout (15 min)	
	2048	4096	91	543	**638**	**0.216**		
	4096	8192	99	591	**690**	**0.442**		
	8192	16384	107	639	**746**	**0.631**		
	16384	32768	115	687	**802**	**1.35**		
	32768	65536	123	735	**858**	**2.92**		
	65536	131072	131	783	**914**	**6.49**		
	131072	262144	Timeout (15 min)					
BV	16	32	29	172	201	0.005	**31**	**0.002**
	32	64	39	233	272	0.006	**63**	**0.004**
	64	128	54	322	376	**0.007**	**127**	0.011
	128	256	76	456	532	**0.010**	**255**	0.040
	256	512	111	668	**779**	**0.014**	799	0.757
	512	1024	173	1039	1212	**0.025**	**1027**	39
	1024	2048	283	1701	**1984**	**0.038**	Timeout (15 min)	
	2048	4096	476	2854	**3330**	**0.067**		
	4096	8192	794	4762	**5556**	**0.120**		
	8192	16384	1337	8024	**9361**	**0.335**		
	16384	32768	2363	14177	**16540**	**0.673**		
	32768	65536	4391	26346	**30737**	**1.42**		
	65536	131072	8395	50372	**58767**	**3.23**		
	131072	262144	16220	97318	**113538**	**8.46**		
	262144	524288	31209	187251	**218460**	**24.44**		
	524288	1048576	58901	353404	**412305**	**75.80**		
	1048576	2097152	Timeout (15 min)					
DJ	16	32	18	90	108	0.006	**18**	**0.001**
	32	64	21	107	128	0.008	**34**	**0.002**
	64	128	24	123	147	**0.008**	**66**	0.038
	128	256	27	139	166	**0.009**	**130**	0.272
	256	512	30	154	**184**	**0.01**	258	2.1
	512	1024	33	170	**203**	**0.011**	516	795.5
	1024	2048	36	186	**222**	**0.014**	Timeout (15 min)	
	2048	4096	39	202	**241**	**0.019**		
	4096	8192	42	218	**260**	**0.028**		
	8192	16384	45	234	**279**	**0.048**		
	16384	32768	48	250	**298**	**0.09**		
	32768	65536	51	266	**317**	**0.182**		
	65536	131072	54	282	**336**	**0.418**		
	131072	262144	57	298	**355**	**0.956**		
	262144	524288	60	314	**374**	**2.57**		
	524288	1048576	63	330	**393**	**7.8**		
	1048576	2097152	66	346	**412**	**26.15**		
	2097152	4194304	69	362	**431**	**95.57**		
	4194304	8388608	72	378	**450**	**180.33**		
	8388608	16777216	Timeout (15 min)					

Table 3.3: Performance of CFLOBDDs against BDDs for increasing number of qubits.

Benchmark	#Qubits	#Boolean Variables	CFLOBDD				BDD	
			#Vertices	#Edges	Total	Time (sec)	#Nodes	Time (sec)
Simon's Alg.	16	64	583	16335	16918	0.71	**5512**	**0.275**
	32	128	123611	14096110	14219721	443.09	**80243**	**3.31**
	64	256	Timeout (90 min)				Timeout (90 min)	
QFT	4	8	7	73	80	0.001	**31**	**0.0001**
	8	16	9	572	581	0.034	**255**	**0.001**
	16	32	15	17868	**17883**	0.128	65535	**0.098**
	32	64	Timeout (15 min)				Timeout (15 min)	
Shor's Alg. $(N, a) = (15, 2)$	4	16	38	338	376	0.09	**69**	**0.04**
Shor's Alg. $(N, a) = (21, 2)$	5	16	72	877	949	2.13	**136**	**0.72**
Shor's Alg. $(N, a) = (39, 2)$	6	16	111	2443	2554	**12.6**	**187**	12.96
Shor's Alg. $(N, a) = (69, 4)$	7	16	176	4331	4487	53.47	**605**	**30.38**
Shor's Alg. $(N, a) = (95, 8)$	7	16	216	4928	5144	53.47	**974**	**41.47**
Shor's Alg. $(N, a) = (119, 2)$	7	16	220	7533	7753	53.47	**3606**	**44.95**
Shor's Alg. $(N, a) = (323, 2)$	9	32	Timeout (15min)				Timeout (15min)	
Grover's Alg.	16	32	17	91	108	**0.009**	**47**	0.214
	32	64	25	138	163	**0.012**	**66**	4.84
	64	128	38	212	**250**	**0.018**	Timeout (15 min)	
	128	256	58	333	**391**	**0.030**		
	256	512	91	531	**622**	**0.080**		
	512	1024	151	886	**1037**	**0.292**		
	1024	2048	259	1535	**1794**	**14.11**		
	2048	4096	450	2674	**3124**	**64.85**		
	4096	8192	766	4569	**5335**	**909.86**		
	8192	16384	Timeout (15 min)					

Table 3.4: Table (cont.) of the performance of CFLOBDDs against BDDs for increasing numbers of qubits.

For GHZ, the algorithms do not depend on an input; the output is solely a function of the number of qubits used. For BV, DJ, QFT, Simon's algorithm, Shor's algorithm, and Grover's algorithm, we ran each algorithm with 50 different randomly selected inputs, for each of the indicated number of qubits. Tabs. 3.3 and 3.4 report the average vertex and average edge counts (for CFLOBDDs), average node count (for BDDs), and average time taken. In the case of Simon's algorithm, CFLOBDDs timed-out on 9 of the 50 test cases, whereas BDDs timed-out on 28 of the 50 test cases; we report the average counts and average times for the test cases that did not time out. BV, DJ, Simon's algorithm, Shor's algorithm, and Grover's algorithm make use of oracles created during a pre-processing step (see also §3.9.1); we do not include the time for oracle construction in the execution time, but we do include it as part of the 15-minute/90-minute timeout threshold. For the case of QFT, the input is one of the basis vectors selected randomly. For 16 qubits and a timeout threshold of 15 minutes, QFT ran to completion in 11 of the 50 runs. The numbers reported

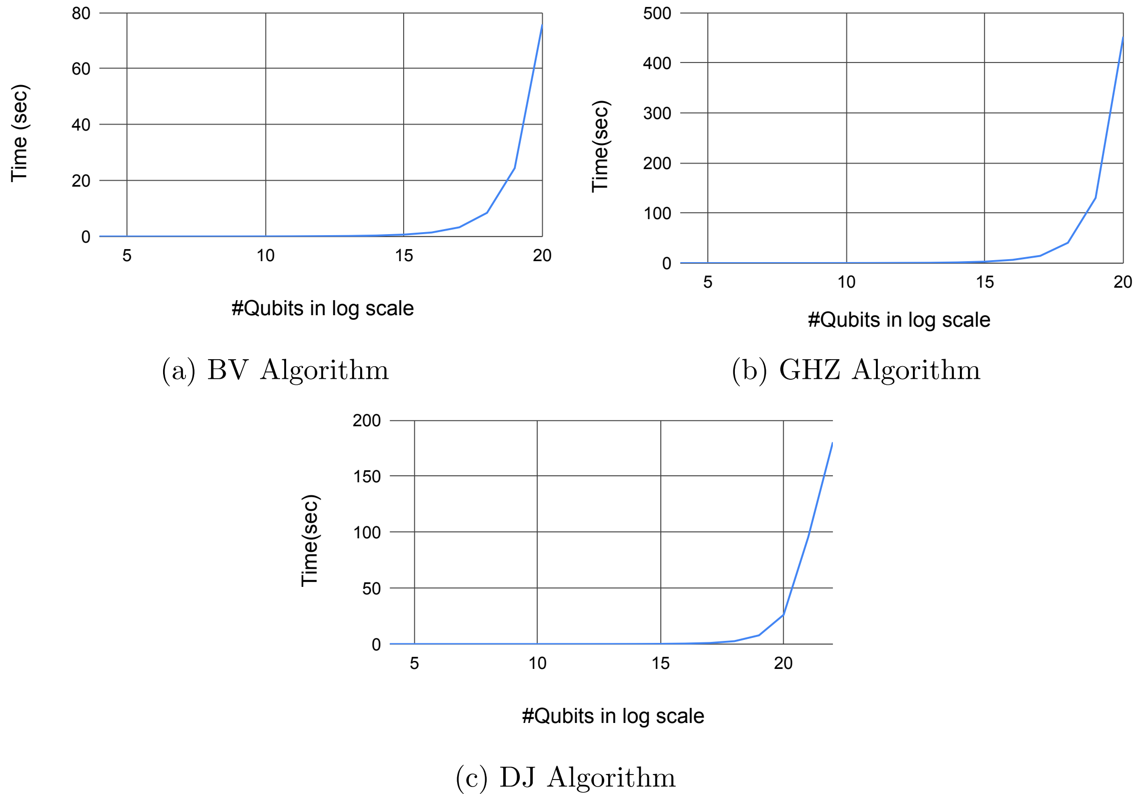
in Tab. 3.4 are the averages for the 11 successful runs. In the entries for Shor's algorithm, N is the number being factored, and a is the value used in the associated "order-finding problem."[24]

In several cases, the problem sizes that completed successfully using CFLOBDDs were dramatically larger than the sizes that completed successfully using BDDs. With a 15-minute timeout, the number of qubits that CFLOBDDs can handle are 65,536 for GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover's Algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

We also ran the CFLOBDD simulations with a 90-minute timeout, both to understand how execution time scales, as a function of number of qubits, and to see how large a problem instance can be handled. Fig. 3.23 shows the time taken (in seconds), with increasing numbers of qubits, for BV, GHZ, and DJ. With a 90-minute timeout, the BV and GHZ algorithms ran to completion with $2^{20} = 1{,}048{,}576$ qubits, and the DJ algorithm ran to completion with $2^{21} = 2{,}097{,}152$ qubits.

For both CFLOBDDs and BDDs, the transition from a problem size that completes successfully to a problem size that fails is rather abrupt. For all of the problems, the time reported for the CFLOBDD run with the largest number of qubits that completes successfully is well under 15 minutes. Unfortunately, for the next larger run, oracle construction timed out after 15 minutes for the BV and DJ algorithms, and as a result we terminated the entire algorithm. For Grover's algorithm, the number of bits for the floating-point representation is 100 for all runs, except for those with 2,048, 4,096, and 8,192 qubits, for which we used 500, 750, and 1,000 bits, respectively. The increased cost of floating-point operations slows down matrix multiplications in Grover's algorithm, causing the 8,192-qubit run to exceed 15 minutes.

[24]Given a, such that $1 < a < N$, the order-finding problem is to find the smallest positive integer r such that $a^r \equiv 1 (mod N)$.



(a) BV Algorithm

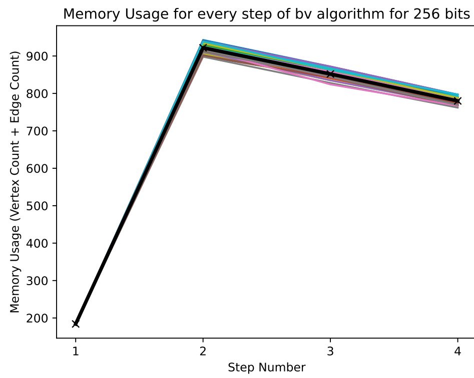
(b) GHZ Algorithm

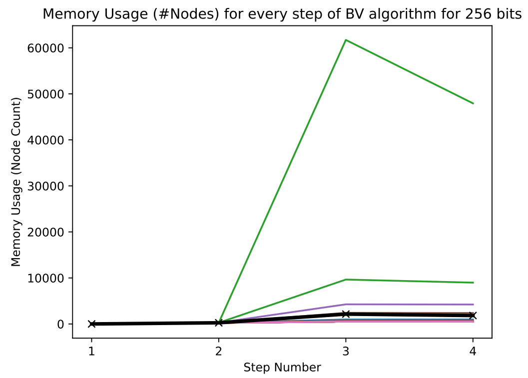
(c) DJ Algorithm

Figure 3.23: CFLOBDD execution time (in seconds) vs. number of qubits (on a log scale) for three of the benchmarks.

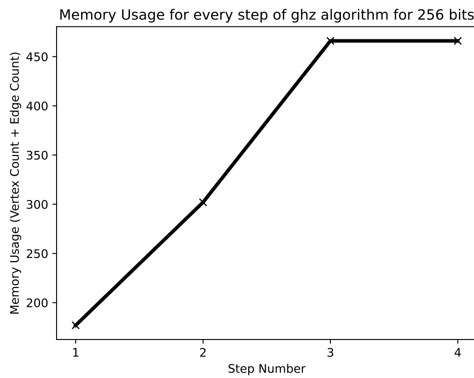
Findings. For smaller numbers of qubits, the more-complex nature of the data structures used in CFLOBDDs resulted in slower execution times than with BDDs. However, CFLOBDDs scaled much better than BDDs as the number of qubits increased, both in terms of memory (i.e., vertices + edges for CFLOBDDs, nodes for BDDs) and execution time. In some cases, the problem sizes that completed successfully using CFLOBDDs were dramatically larger than the sizes that completed successfully using BDDs. In particular, the number of qubits that could be handled using CFLOBDDs was larger—compared to BDDs—by a factor of 128× for GHZ; 1,024× for BV; 8,192× for DJ; and 128× for Grover's algorithm.

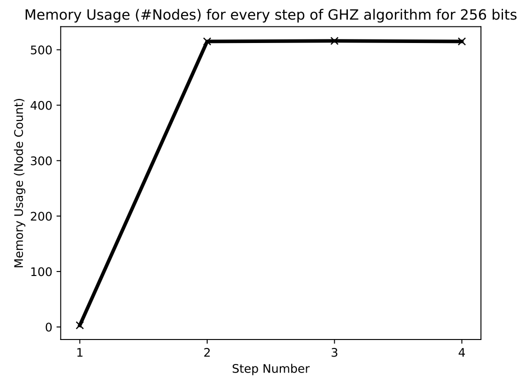
Intermediate swell. In many of the algorithms, the initial and final CFLOBDD and BDD structures are of reasonable size, but there is an intermediate swell in size as the algorithm runs. Figs. 3.24 and 3.25 show how size evolves in the various steps of



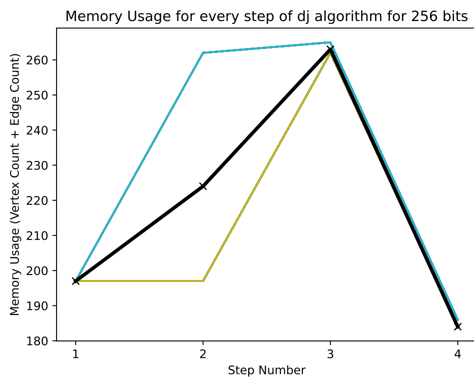


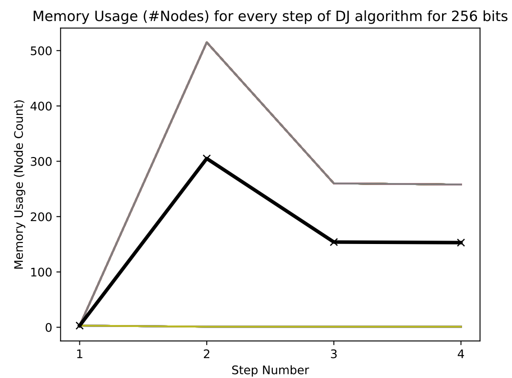
(a) BV algorithm 256 qubits





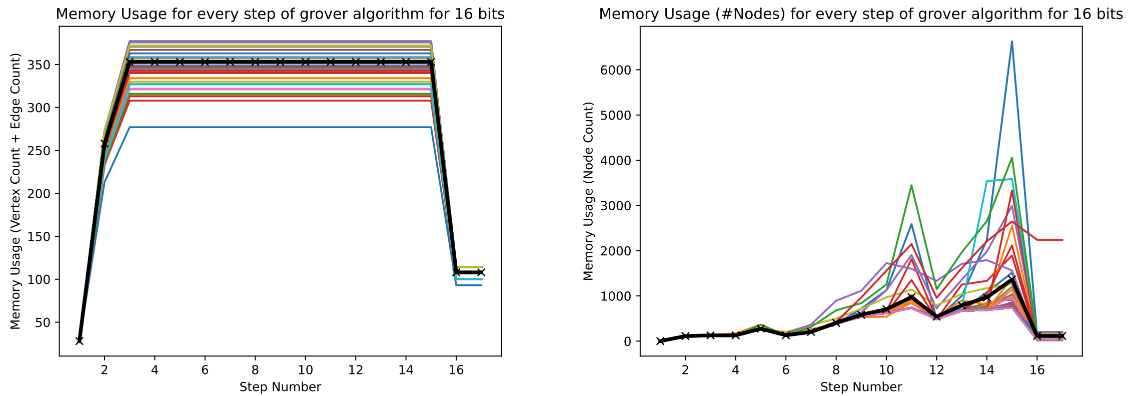
(b) GHZ algorithm 256 qubits



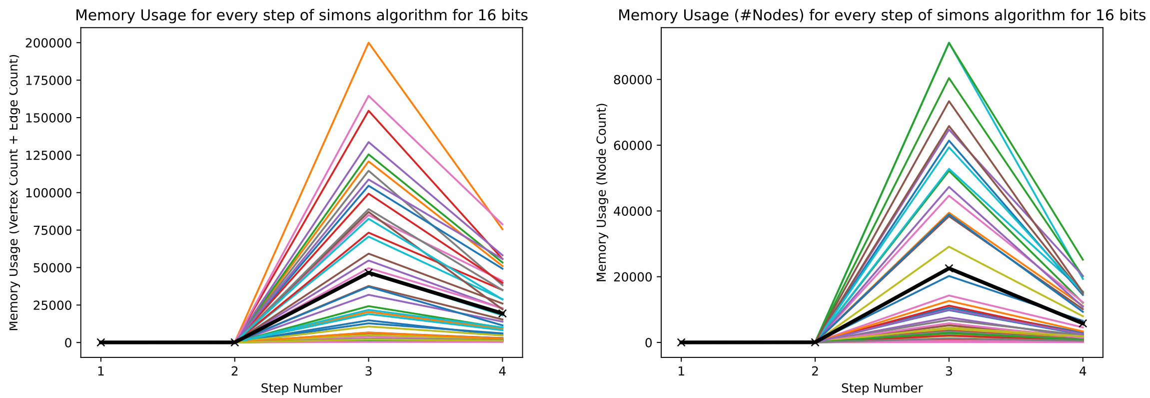


(c) DJ algorithm 256 qubits

Figure 3.24: Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.)



(a) Grover's algorithm 16 qubits



(b) Simon's algorithm 16 qubits

Figure 3.25: Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.)

five of the algorithms during the CFLOBDD-based and BDD-based simulations. The figures show how size evolves for all 50 runs, along with the average value at every step (highlighted in black). Fig. 3.25a shows that the CFLOBDD simulation of Grover's algorithm uses constant space from steps 3 to 15. The explanation is that, although the state vector changes at each step, the size of the CFLOBDD representation of the state vector does not change.

Comparison with Tensor Networks. We also compared the performance of CFLOBDDs with Quimb Gray (2018), a state-of-the-art quantum simulator. Tabs. 3.5 and 3.6 shows the performance of our CFLOBDD implementation and Quimb on the previously discussed quantum benchmarks. For the Quimb-based simulations of GHZ, BV, DJ, and Grover's algorithm, we used Matrix Product States (MPSs) Vidal (2003); Banuls et al. (2009) and Matrix Product Operators (MPOs) Verstraete et al. (2004) in algorithms modeled after the ones described in Woolfe (2015). For Simon's algorithm, we noticed that directly creating a circuit and performing contraction using Quimb led to better scalability than using MPS/MPOs. For QFT, we tried both the standard circuit Nielsen and Chuang (2000) and the nearest-neighbor circuit mentioned in Fowler et al. (2004). We found that the Quimb-based simulation results for both circuits are very similar, and only the former are reported here. For Shor's algorithm, we use the $2n + 3$ circuit from Beauregard (2002), but the internal gates are created directly, as mentioned in Woolfe (2015) (and hence the circuit only has $2n + 1$ qubits). For Grover's algorithm, we found that the maximum number of qubits that can be simulated using Quimb with a 15-minute timeout is 29 qubits.[25]

These experiments show that, on some of the benchmarks, CFLOBDDs scale to much larger problem sizes than the Quimb tensor-network package, but on other benchmarks Quimb performs much better than CFLOBDDs.

What allows CFLOBDDs to perform so well on Grover's algorithm? In each run of the CFLOBDD simulation of Grover's algorithm, a random 4096-bit string s is chosen, then the Grover oracle matrix is constructed, along with the Grover diffusion operation, which are then multiplied together. A version of Grover's algorithm based on repeated squaring of the product matrix is carried out (via operations that use the cumulative-product matrix—which depends on s—but the operations are

[25] With 32 qubits, Quimb takes 1496.6sec $\approx$ 25min.

Benchmark	#Qubits	CFLOBDD (Time in sec)	Quimb (Time in sec)
GHZ	16	**0.005**	0.222
	32	**0.007**	0.644
	64	**0.010**	2.29
	128	**0.015**	9.23
	256	**0.027**	40.31
	512	**0.051**	191.77
	1024	**0.107**	Timeout (15 min)
	⋮	⋮	
	65536	**6.49**	
	131072	Timeout (15 min)	
BV	16	**0.005**	0.264
	32	**0.006**	0.773
	64	**0.007**	2.75
	128	**0.010**	11.08
	256	**0.014**	49.49
	512	**0.025**	243.69
	1024	**0.038**	Timeout (15 min)
	⋮	⋮	
	524288	**75.80**	
	1048576	Timeout (15 min)	
DJ	16	**0.006**	0.256
	32	**0.008**	0.761
	64	**0.008**	2.75
	128	**0.009**	11.18
	256	**0.010**	49.33
	512	**0.011**	243.01
	1024	**0.014**	Timeout (15 min)
	⋮	⋮	
	4194304	**180.33**	
	8388608	Timeout (15 min)	
Simon's Alg.	16	**0.71**	2.56
	32	443.09	**17.34**
	64	Timeout (15 min)	**267**
	128		Timeout (15min)

Table 3.5: Performance of CFLOBDDs against Quimb on quantum benchmarks for different numbers of qubits.

Benchmark	#Qubits	CFLOBDD (Time in sec)	Quimb (Time in sec)
QFT	4	**0.001**	0.023
	8	**0.034**	0.035
	16	0.128	**0.074**
	32	Timeout (15 min)	**0.231**
	64		**1.64**
	128		**10.32**
	256		**103.65**
	512		Timeout (15min)
Shor's Alg. $(N, a) = (15, 2)$	4	0.09	**0.08**
Shor's Alg. $(N, a) = (21, 2)$	5	2.13	**0.1**
Shor's Alg. $(N, a) = (39, 2)$	6	12.6	**0.11**
Shor's Alg. $(N, a) = (69, 4)$	7	53.47	**0.12**
Shor's Alg. $(N, a) = (95, 8)$	7	42.8	**0.12**
Shor's Alg. $(N, a) = (119, 2)$	7	64.8	**0.12**
Shor's Alg. $(N, a) = (323, 2)$	9	Timeout (15 min)	**0.27**
⋮	⋮		⋮
Shor's Alg. $(N, a) = (6085, 8)$	12		**107.28**
Shor's Alg. $(N, a) = (11611, 2)$	13		Out of Memory
Grover's Alg.	16	**0.009**	3.26
	32	**0.012**	Timeout (15 min)
	⋮	⋮	
	4096	**909.86**	
	8192	Timeout (15 min)	

Table 3.6: Table (cont.) of the performance of CFLOBDDs against Quimb on quantum benchmarks for different numbers of qubits.

oblivious to the value of s itself); the algorithm's answer s' is retrieved; and finally s and s' are compared to make sure that the computed result is correct.

The reason that this process is space-efficient is that the Grover oracle is basically a "-1 hot encoding" of s, and thus can be constructed by an algorithm that is a mixture of the principles used in the algorithms for constructing the representations of (i) projection functions (§3.6.1.2), and (ii) the identity matrix (§3.8.1.2). In the largest cases of Grover's algorithm that completed successfully within 15 minutes, the matrix has dimensions $2^{4096} \times 2^{4096}$; all off-diagonal entries are 0; and all diagonal entries are 1 except for the (s, s) entry, which is -1. To represent this matrix, one needs $8{,}192 = 2^{13}$ Boolean variables: 4,096 for the row-index and 4,096 for the column-index.

Benchmark	#Vars.	CFLOBDD				BDD	
		#Vertices	#Edges	Size	Time (sec)	Size	Time (sec)
c17	5	13	54	67	**0.001**	**22**	0.004
c432	36	2503	44328	46831	0.62	**27916**	**0.01**
c880	60	1481	13679	**15160**	0.07	15478	**0.01**
c6288_8	16	1974	22966	24940	1.07	**18516**	**0.02**
c6288_9	18	6319	118376	124695	3.86	**53416**	**0.06**
c6288_10	20	6998	409314	416312	7.72	**150898**	**0.25**
c6288_11	22	59110	1107573	1166683	25.23	**428190**	**0.97**
c6288_12	24	326924	3838786	4165710	357.02	**1219808**	**3.64**

Table 3.7: Table of the performance of CFLOBDDs against BDDs on different combinatorial circuits.

There is a CFLOBDD representation of this matrix whose highest-level grouping is at level 13—thus, there are 14 levels in total, counting level 0. Moreover, the CFLOBDD has only a constant number of groupings at each of the 14 levels, so the matrix is one for which the CFLOBDD representation exhibits double-exponential compression.

Although multiplication of matrices represented by CFLOBDDs is not particularly efficient (see the last row of Tab. 3.1), there is little or no infill caused by the repeated-squaring operations, and so the matrix representation has only a limited amount of intermediate swell. (See the left-hand graph in Fig. 3.25(a) for a plot of memory usage for the CFLOBDD implementation of Grover's algorithm for 16 qubits.)

3.9.2.3 RQ3: Do CFLOBDDs outperform BDDs when used in hardware verification benchmarks (in terms of time and space)?

We also evaluated CFLOBDDs and BDDs in terms of time and space when run on hardware-verification benchmarks – `iscas85` – consisting primarily of combinatorial circuits. Tab. 3.7 shows the performance of CFLOBDDs against BDDs. We observe that in this case, BDDs outperform CFLOBDDs on most benchmarks

in terms of size [26] and running time. This experiment shows that the combinatorial circuits either (1) do not have enough "symmetry" that can be used by the hierarchy exploited by CFLOBDDs, or (2) the "symmetry" in the circuits does not fit well with the balanced grammar hierarchy of CFLOBDDs. The latter question is addressed in experiments presented in §7.4.1, where we introduce a CFLOBDD variant in which the grammar for the hierarchy is specifiable by the client.

The experiment shows that when Boolean functions do not have enough of a "repeated" structure, then using BDDs to represent the functions is a better choice than CFLOBDDs.

[26]We consider the size of a BDD to be 2*#nodes, because every BDD node has two pointers to other BDDs.

Chapter 4: WCFLOBDDs: Weighted Context-Free-Language Ordered Binary Decision Diagrams

4.1 Introduction

As discussed in Chapter 2, BDDs are a data structure that provides (i) a canonical representation of Boolean functions, relations, matrices, etc., and (ii) algorithms for operating on them in compressed form. Likewise, in the previous chapter Chapter 3, we discussed CFLOBDDs, another data structure that provide exponential compression over BDDs. BDDs were originally designed for Boolean-valued functions, and failed to address the closely related functions: $\{0,1\}^n \rightarrow D$, where D is a set. Multi-terminal BDDs (MTBDDs) Fujita et al. (1997) (or ADDs Bahar et al. (1997)) are a relatively straightforward BDD extension for such functions, but have their own drawback: for a function h whose range has a large number of different values V ($V \subseteq D$), the size of an MTBDD (or a multi-terminal CFLOBDD) for h cannot be any smaller than $|V|$. To date, the most successful solution to this issue has been *Weighted BDDs* (WBDDs)—BDD-like structures with weights on edges Niemann et al. (2016); Viamontes et al. (2004). For a given assignment a to h's argument variables, $h(a)$ is computed by "multiplying" the weights on the path for a. This approach yields succinct representations when factors of the values in V can be reused in the WBDD data structure.

Consider two types of functions – (i) Exponential function EXP_n: $\{0,1\}^n \rightarrow \{2^j \mid j \in \{0 \ldots 2^n{-}1\}\}$, where there are 2^n unique values and thereby having weights on edges could provide more compression, and (ii) Equality function with n variables $EQ_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \rightarrow \{0,1\}$; though the output has only two values, an efficient and more compressed representation is possible by reusing substructures.

To represent these two types of functions efficiently, we introduce a new data

structure called *Weighted Context-Free-Language Binary Decision Diagrams* (WCFLOBDDs). The data structure borrows ideas from WBDDs, and CFLOBDDs to efficiently represent a broader set of functions than possible heretofore.

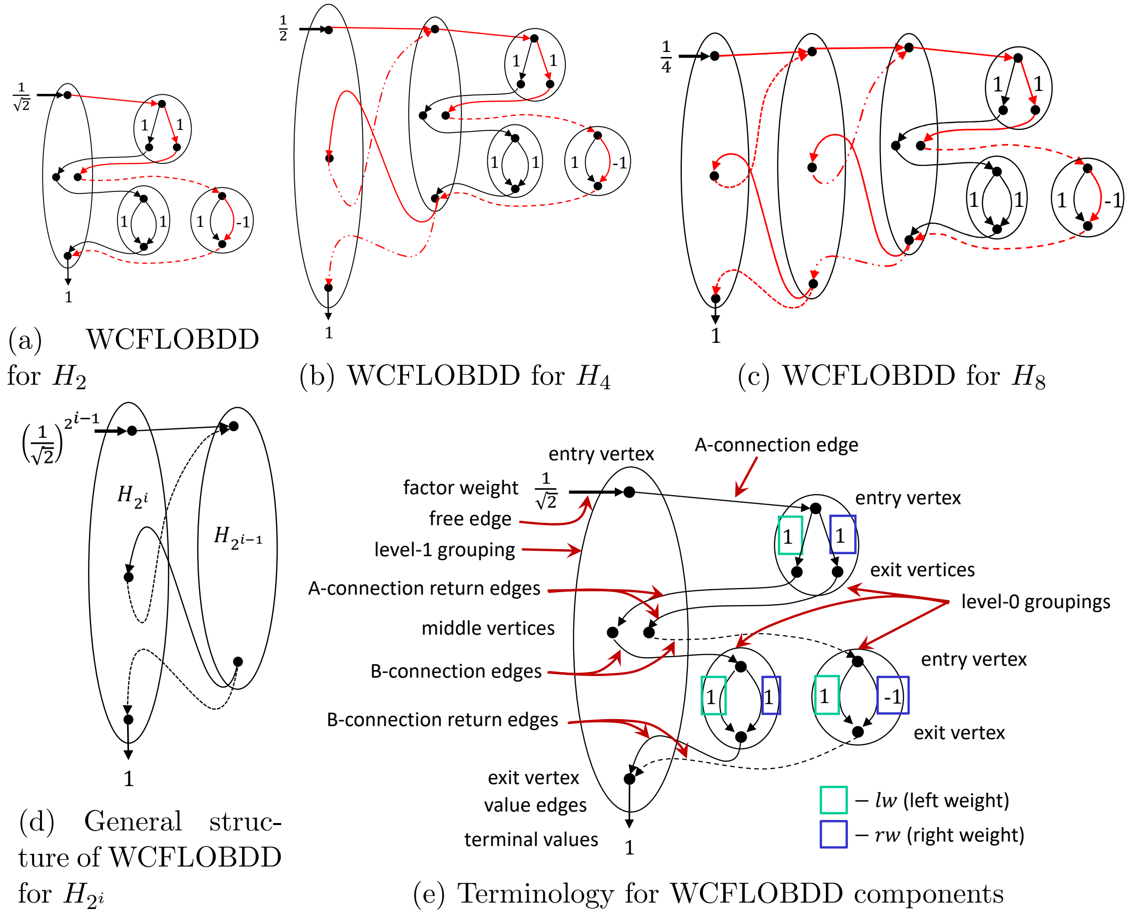
We show that, in the best case, WCFLOBDDs provide an exponential factor of succinctness over WBDDs and CFLOBDDs. The following diagram shows the design space of BDDs, WBDDs, CFLOBDDs, and WCFLOBDDs:

		Hierarchical	
		no	yes
Weights	no	BDD *Bryant* (1986) *Fujitaet al.* (1997) *Baharet al.* (1997)	CFLOBDD 3
	yes	WBDD *Niemannet al.* (2016) *Viamonteset al.* (2004)	WCFLOBDD (this chapter)

We evaluated WCFLOBDDs, WBDDs, and CFLOBDDs on both synthetic and quantum-simulation benchmarks, and found that WCFLOBDDs perform better than WBDDs and CFLOBDDs on most benchmarks. On the quantum-simulation benchmarks, with a 15-minute timeout, the number of qubits that can be handled by WCFLOBDDs is 1,048,576 for GHZ (1× over CFLOBDDs, 256× over WBDDs); 262,144 for BV and DJ (2× over CFLOBDDs, 64× over WBDDs); and 2,048 for QFT (128× over CFLOBDDs, 2× over WBDDs). Such results support the conclusion that, at least for some applications, WCFLOBDDs permit much larger problem instances to be handled than was possible previously.

Organization. The rest of the chapter is organized as follows:

- We define WCFLOBDDs, a new data structure to represent Boolean functions, relations, matrices, etc. (§4.2).
- We give *algorithms* for WCFLOBDD operations (§4.3).



(a) WCFLOBDD for H_2

(b) WCFLOBDD for H_4

(c) WCFLOBDD for H_8

(d) General structure of WCFLOBDD for H_{2^i}

(e) Terminology for WCFLOBDD components

Figure 4.1: (a), (b), and (c) show WCFLOBDDs for the first three matrices in $\mathcal{H}$—H_2, H_4, and H_8—with the variable ordering $\langle x_0, y_0, x_1, y_1, \ldots \rangle$ ($\vec{x}$: row; $\vec{y}$: column). (d) shows the general structure of a WCFLOBDD that represents H_{2^i}. (e) illustrates the constituents of the WCFLOBDD for H_2.

- We show exponential gaps between (i) WCFLOBDDs and CFLOBDDs, and (ii) WCFLOBDDs and WBDDs (§4.2.3).

- We find that WCFLOBDDs perform as well as or better than WBDDs and CFLOBDDs on most synthetic and quantum-simulation benchmarks (§4.4).

4.2 Weighted CFLOBDDs (WCFLOBDDs)

4.2.1 Basic Structure of WCFLOBDDs

The hierarchy in a WCFLOBDD is similar to that in a CFLOBDD. The number of variables at each level is always a power of 2, and at each level k, the variables are divided in half—the A call interpreting $x_0, \ldots, x_{2^{k-1}-1}$, and the B calls interpreting $x_{2^{k-1}}, \ldots, x_{2^k-1}$. The main difference from a CFLOBDD is that at level 0, each transition has a *weight*. This approach produces a structure that is akin to a non-recursive, single-entry, multi-exit, *weighted* hierarchical finite-state machine.

We define WCFLOBDDs in two parts. Defn. 4.1 defines the basic structure of WCFLOBDDs, whose various elements are depicted in Fig. 4.1(e). Defn. 4.5 imposes some additional structural invariants to ensure that WCFLOBDDs provide a canonical representation of Boolean functions. Where necessary, we distinguish between *mock-WCFLOBDDs* (Defn. 4.1) and *WCFLOBDDs* (Defn. 4.5), although we typically drop the qualifier "mock-" when there is little danger of confusion.

Definition 4.1 Mock-WCFLOBDD; see Fig. 4.1(e). A *mock-WCFLOBDD* at level k is a hierarchical structure made up of some number of *groupings*, of which there is one grouping at level k, called the *head-grouping*, and at least one grouping at each level $0, 1, \ldots, k-1$. A grouping is a collection of vertices and edges (to other groupings), and weights (for level-0 groupings), with the structure described below.

A mock-WCFLOBDD is a triple $\langle f, G, V \rangle$, where $f \in \mathcal{D}$ is the *factor weight* of the *free-edge* (defined below), G is the head-grouping (conceptually "topmost;" portrayed leftmost), and $V \in \{[\bar{0}], [\bar{1}], [\bar{0}, \bar{1}], [\bar{1}, \bar{0}]\}$—where "$[\cdot]$" denotes a tuple—is the sequence of *terminal values*, whose cardinality must match the number of exit vertices of G (defined below).

- *Groupings:* Each oval-shaped object is called a *grouping*, and every grouping is associated with a level l (≥ 0). Each grouping at level i is always connected to groupings at level i-1.

- *Vertices:* Each grouping g at a level ≥ 1 has a unique *entry vertex* (the dot at the top of g), and some *middle vertices* and *exit vertices* (dots in the middle and at the bottom of g). The three sets of vertices are disjoint. (It is useful to think of the collections of middle vertices and exit vertices as two sequences, each numbered from left-to-right.)

  A level-0 grouping (conceptually "bottommost;" portrayed to the right) has a unique entry vertex, no middle vertices, and two edges (the 0-edge and 1-edge), which are directed to either one or two exit vertices. There are two kinds of level-0 groupings: *fork groupings* (with two exit vertices) and *don't-care groupings* (with one exit vertex).

- *A-connection:* The grouping between g's entry and middle vertices is called the A-connection. The edge from g's entry vertex is called an A-connection edge, and the edges from the A-connection's exit vertices to g's middle vertices are called *A-connection return edges.*

- *B-connections:* The groupings between g's middle and exit vertices are called B-connections. The edges from g's middle vertices are called B-connection edges and the edges from a B-connection's exit vertices to g's exit vertices are called *B-connection return edges.*

  There is one A-connection, but there can be more than one B-connection (each with a set of B-connection return edges back to g's exit vertices).

- *Value Edges:* The edges that connect the exit vertices of the head-grouping to the terminal values V are called *value edges.*

The only edges that are assigned weights are the free-edge and the edges of level-0 groupings.

- The incoming edge to the entry vertex of the outermost grouping is called the *free-edge.* The weight $f \in \mathcal{D}$ associated with the free-edge is the *factor weight.*

- A level-0 grouping has a pair of semi-field weights (lw, rw): $lw \in \mathcal{D}$ is the weight for the decision-edge for 0 (false); $rw \in \mathcal{D}$ is the weight for the decision-edge for 1 (true).

The terminal values connected to the exit vertices of the head-grouping can only be $\bar{0}, \bar{1} \in \mathcal{D}$. For a given function h, the appropriate sequence of terminal values V can be understood by considering the decision tree for h:

- $[\bar{0}]$ only occurs for the function $\lambda\vec{x}.\bar{0}$. All leaves of the decision tree are $\bar{0}$.
- $[\bar{1}]$ occurs for a function for which the value is never $\bar{0}$ (for any assignment to the Boolean variables). No leaf of the decision tree is $\bar{0}$.
- $[\bar{1}, \bar{0}]$ occurs when a function has the value $\bar{0}$ for at least one assignment, but the leftmost leaf of the decision tree is non-$\bar{0}$.
- $[\bar{0}, \bar{1}]$ occurs when the function has a non-$\bar{0}$ value for at least one assignment, and the leftmost leaf of the decision tree is $\bar{0}$.

Fig. 4.1(a) shows the WCFLOBDD for H_2. It consists of 4 groupings, of which 3 are at level 0 and one is at level 1; the upper-right level-0 grouping is a fork grouping, and the bottom-right level-0 groupings are don't-care groupings.

A grouping at level-k encodes 2^k variables and 2^{2^k} paths. As seen in Fig. 4.1(a), the grouping at level-1 encodes two variables, and therefore 4 paths. In Fig. 4.1(b), the largest oval is a level-2 grouping, which encodes 4 variables: 2 variables through the A-connection grouping and 2 variables through the B-connection grouping, and therefore 2^4 paths. In general, at level k the 2^k variables are divided into halves: 2^{k-1} in the A-connection and 2^{k-1} in the B-connection.

Fig. 4.2 shows an object diagram of the WCFLOBDD for matrix H_2 from Fig. 4.1(a). In this representation, which will be used in all algorithms stated in the paper, there are no explicit entry, middle, and exit vertices. The pointer to the

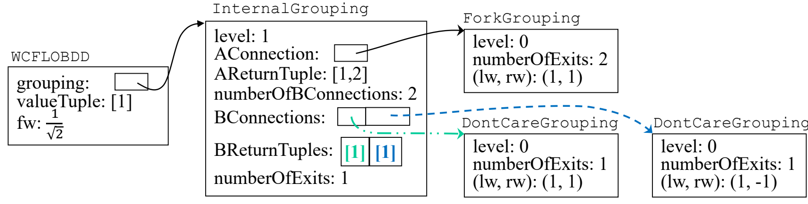


Figure 4.2: Object diagram of the WCFLOBDD for matrix H_2 (Fig. 4.1(a)).

object serves as the entry vertex; numbers in [1..*numberOfBConnections*] stand for the middle vertices; and those in [1..*numberOfExits*] stand for the exit vertices. (The context determines whether an index is being used as a middle vertex or exit vertex.) As can be seen by comparing Fig. 4.2 to Fig. 4.1(a), the various edges that appear in Fig. 4.1(a) are stored as pointers to Groupings and elements of ReturnTuples. For a given A-connection edge or B-connection edge c from grouping g_i to g_{i-1}, the associated ReturnTuple rt_c consists of the sequence of targets of return edges from exit vertices of g_{i-1} to the middle or exit vertices of g_i that correspond to c. There are three grouping classes: *InternalGrouping*, *ForkGrouping*, and *DontCareGrouping*. The latter two are level-0 groupings, which store left and right edge weights (lw, rw); thus, operations that construct a level-0 grouping take two inputs, lw and rw. All groupings at levels ≥ 1 are *InternalGrouping*s.

To be able to make inductive arguments about WCFLOBDDs, we define

Definition 4.2 Mock-proto-WCFLOBDD**.** A *mock-proto-WCFLOBDD* at level i is a grouping at level i, together with the lower-level groupings to which it is connected (and the connecting edges). In other words, a mock-proto-WCFLOBDD has the following recursive structure:

- a mock-proto-WCFLOBDD at level 0 is either a fork grouping or a don't-care grouping
- a mock-proto-WCFLOBDD at level i is headed by a grouping at level i whose

– A-connection edge and associated return edges "call" a level-(i-1) mock-proto-WCFLOBDD

– B-connection edges and their associated return edges "call" some number of level-(i-1) mock-proto-WCFLOBDDs.

A mock-WCFLOBDD is a mock-proto-WCFLOBDD in which (i) the exit vertices of the mock-proto-CFLOBDD have been associated with specific terminal values V, and (ii) a factor weight has been supplied. (In other words, in the tuple $\langle f, G, V \rangle$ in Defn. 4.1, G serves double-duty: both as the head-grouping, and as the mock-proto-WCFLOBDD of which it is the head-grouping.)

When interpreting a Boolean assignment, the path taken must abide by the following principle (similar to that of CFLOBDDs):

Matched-Path Principle. *When a path follows an edge that returns to level i from level $i-1$, it must follow an edge that matches the closest preceding edge from level i to level $i-1$.*

Only *matched*-paths that start at the entry vertex of the WCFLOBDD's highest-level grouping and end at a terminal value are considered in interpreting a WCFLOBDD.

Decisions are taken only at the level-0 groupings: the edges in a level-0 grouping correspond to the two choices for a Boolean variable. A grouping can be reused multiple times at different "calls," as seen in Fig. 4.1(b)–(d). Because of sharing at all levels, a given level-0 grouping (decision grouping) does not represent a specific variable, according to the following principle (again, similar to that of CFLOBDDs):

Contextual-Interpretation Principle. *A level-0 grouping is not associated with a specific Boolean variable. Instead, the variable that a level-0 grouping refers to is determined by the context: the jth level-0 grouping visited along a matched path is used to interpret the jth Boolean variable.*

Function evaluation. As with BDDs, it is often useful to think of an assignment a as a sequence of Boolean values. We use "$||$" to denote concatenation of two half-size sequences: $a = a_1||a_2$, where $|a_1| = |a_2| = |a|/2$ (and $|a|$ is a power of 2).

Definition 4.3 Exit vertex reached via assignment a**.** Suppose that g is a grouping. Let $g.BReturnTuples[j]$ be the sequence of return edges for the jth B-connection of g; i.e., $g.BReturnTuples[j][k]$ represents the exit vertex of g that is connected to the kth exit vertex of the jth B-connection of g. For a grouping g and an assignment a, $\langle g\rangle(a)$ denotes the exit vertex of g reached via assignment a, defined as follows:

$$\langle DontCareGrouping\rangle([0]) \stackrel{\text{def}}{=} 1 \qquad \langle DontCareGrouping\rangle([1]) \stackrel{\text{def}}{=} 1$$
$$\langle ForkGrouping\rangle([0]) \stackrel{\text{def}}{=} 1 \qquad \langle ForkGrouping\rangle([1]) \stackrel{\text{def}}{=} 2$$
$$\langle g\rangle(a) \stackrel{\text{def}}{=} g.BReturnTuples[j][k], \text{where } g.level \geq 1, a = a_1||a_2,$$
$$j = \langle g.AConnection\rangle(a_1), \text{ and } k = \langle g.BConnections[j]\rangle(a_2).$$

Definition 4.4 WCFLOBDD evaluation**.** Given WCFLOBDD $C = \langle f, G, V\rangle$ for $h : \{0,1\}^n \rightarrow \mathcal{D}$, and an assignment a for h's arguments, the value $h(a)$ is obtained from C, denoted by $[\![C]\!](a)$, as the product of the edge weights on the path for a in C:

$$[\![C]\!](a) \stackrel{\text{def}}{=} f \cdot [\![G]\!](a) \cdot V[\langle G\rangle(a)]$$
$$[\![G]\!](a) \stackrel{\text{def}}{=} [\![G.AConnection]\!](a_1) \cdot [\![G.BConnections[j]]\!](a_2),$$

where $a = a_1||a_2$ and $j = \langle G.AConnection\rangle(a_1)$.

In the case of Fig. 4.1(a), $\mathcal{D} = \langle \mathbb{R}, +, \times, 0, 1\rangle$. For $a = \{x_0 \mapsto 1, y_0 \mapsto 1\}$, the path highlighted in bold evaluates to $H[1][1] = \frac{1}{\sqrt{2}} \times 1 \times -1 \times 1 = \frac{-1}{\sqrt{2}}$, as desired. Fig. 4.1(b) and Fig. 4.1(c) show the WCFLOBDD representations of H_4 and H_8. Fig. 4.1(d) shows the generic structure for how H_{2^i} is formed from $H_{2^{i-1}}$. As one can see, the grouping at level i-1 is shared twice at each level $i \geq 2$, giving a representation that is exponentially more succinct than a WBDD.

Intuition for succinctness: Consider the WBDD and WCFLOBDD for H_4 in Fig. 2.3(b) and Fig. 4.1(b), respectively. The A-connection of the level-2 grouping in Fig. 4.1(b) represents the top half of the WBDD structure (for variables $\langle x_0, y_0\rangle$),

and the B-connection of the level-2 grouping represents the bottom half of the WBDD structure (for variables $\langle x_1, y_1 \rangle$). However, because the A-connection grouping and B-connection grouping of the WCFLOBDD have the identical structure, they are shared, producing a more succinct representation.

Implementation. Our WCFLOBDD implementation will be made available in open-source form if the paper is accepted. The implementation is parameterized—via C++ templatization—to work with any user-defined semi-field $\mathcal{D}$ (and its corresponding operations $+$ and $\times$).

4.2.2 Canonicity

Like other decision diagrams, we ensure *canonicity*—every function has a unique representation—by imposing certain structural invariants (Wegener, 2000, p. 48), §3.3.1.

Intuition for canonicity: The structural invariants are designed to ensure that—for a given order on the Boolean variables—each Boolean function has a unique, canonical representation as a WCFLOBDD. In reading Defn. 4.5 below, it will help to keep in mind that the goal of the invariants is to force there to be a *unique* way to fold a given decision tree into a WCFLOBDD that represents the same Boolean function. The decision-tree folding method is discussed later in this section and in §L, and is illustrated in Fig. 4.3, but the main characteristic of the folding method is that it works *greedily, left to right*, similar to that of CFLOBDDs. This directional bias shows up in structural invariants 1, 2a, and 2b.

Definition 4.5. A *(proto-)WCFLOBDD* is a mock-(proto-)WCFLOBDD in which every grouping/proto-WCFLOBDD satisfies the *structural invariants* given below. (See Fig. 3.6.)

Structural Invariants. The structural invariants mainly deal with (i) the organization of return tuples, and (ii) weights in level-0 groupings. The following invariants

deal with constraints on return tuples. Let c be an A-connection or B-connection edge with return tuple rt_c from g_{i-1} to g_i.

1. If c is an A-connection edge, then rt_c must map g_{i-1}'s exit vertices 1-to-1, in order, onto g_i's middle vertices.

2. If c is the B-connection edge whose source is middle vertex $j+1$ of g_i and whose target is g_{i-1}, then rt_c must meet two conditions:

   (a) It must map g_{i-1}'s exit vertices 1-to-1 (but not necessarily onto) g_i's exit vertices.

   (b) It must "compactly extend" the set of exit vertices in g_i defined by the return tuples for the previous j B-connections (similar to condition (1) on A-connections): Let rt_{c_1}, rt_{c_2}, ..., rt_{c_j} be the return tuples for the first j B-connection edges out of g_i. Let S be the set of indices of exit vertices of g_i that occur in return tuples rt_{c_1}, rt_{c_2}, ..., rt_{c_j}, and let n be the largest value in S. (That is, n is the index of the rightmost exit vertex of g_i that is a target of any of the return tuples rt_{c_1}, rt_{c_2}, ..., rt_{c_j}.) If S is empty, then let n be 0.

      Now consider rt_c ($= rt_{c_{j+1}}$). Let R be the (not necessarily contiguous) sub-sequence of rt_c whose values are strictly greater than n. Let m be the size of R. Then R must be exactly the sequence $[n+1, n+2, \ldots, n+m]$.

3. A proto-WCFLOBDD can be used (i.e., pointed to by other higher-level groupings) multiple times, but a proto-WCFLOBDD never contains two separate instances of equal proto-WCFLOBDDs. (Equality is defined inductively on the hierarchical structure of groupings.)

4. For every pair of B-connection edges c and c' of grouping g_i, with associated return tuples rt_c and $rt_{c'}$, if c and c' lead to two level i-1 proto-WCFLOBDDs, say p_{i-1} and p'_{i-1}, such that $p_{i-1} = p'_{i-1}$, then the associated return tuples

must be different (i.e., $rt_c \neq rt_{c'}$). This condition means that two B-connection edges c and c' can "call" the same proto-WCFLOBDD, but the two sets of return edges rt_c and $rt_{c'}$ have to be different.

The following invariants pertain to the weights on edges in level-0 groupings:

5. The left-edge and right-edge weights (lw, rw) must obey:

$$(lw, rw) = \begin{cases} (\bar{1}, rw) & \text{when } lw \neq \bar{0} \\ (\bar{0}, \bar{1}) & \text{otherwise} \end{cases}$$

6. If a middle vertex m of g_i has a path with aggregated weight $\bar{0}$ from the entry vertex of g_i to m, then there must exist an exit-vertex e such that every level-0 grouping edge on every path from m to e has weight $\bar{0}$.

7. If e is an exit vertex of the outermost grouping such that all paths of weight $\bar{0}$ lead to e, then e must map to the terminal value $\bar{0}$.

8. If all paths lead to the terminal value $\bar{0}$, then the factor-weight must be $\bar{0}$.

Finally, in a level-k *WCFLOBDD* $\langle f, G, V \rangle$,

9. The head-grouping G of $\langle f, G, V \rangle$ heads a level-k proto-WCFLOBDD.

10. Value-tuple V is one of $\{[\bar{0}], [\bar{1}], [\bar{0}, \bar{1}], [\bar{1}, \bar{0}]\}$, and thus maps each exit vertex of G to a distinct terminal value. If there exists an exit vertex e of G such that all paths leading to e have weight $\bar{0}$, then e is mapped to $\bar{0}$.

Canonicity Theorem Now we state the canonicity theorem for WCFLOBDDs:

Theorem 4.1. If C_1 and C_2 are two level-k WCFLOBDDs for the same Boolean function, and use the same variable ordering, then C_1 and C_2 are isomorphic.

Proof Sketch. We prove the theorem by establishing three properties:

1. Every level-k WCFLOBDD represents a decision tree (for the same Boolean function with the same variable ordering) with 2^{2^k} leaves.

2. Every decision tree with 2^{2^k} leaves is represented by some level-k WCFLOBDD.

3. No decision tree with 2^{2^k} leaves is represented by more than one level-k WCFLOBDD (up to isomorphism).

Obligation 1. This property is established by induction. Let $P(l)$ be the number of leaves of a decision tree for the function represented by a level-l proto-WCFLOBDD. We want to prove that $P(l) = 2^{2^l}$.

- Base Case: $l = 0$, #variables = 1, #paths = #leaves in a decision tree = 2.

- Induction Step: Let $P(m) = 2^{2^m}$. We wish to prove that $P(m+1) = 2^{2^{m+1}}$. Given a level-l proto-WCFLOBDD with outermost grouping g_l, by construction the number of variables of g_l is equally divided between the A-connection and B-connections of g_l. Hence, for a level-m+1 proto-WCFLOBDD, we have $P(m+1) = \sum_j A_j(m) \cdot P(m) = (\sum_j A_j(m)) \cdot P(m) = P(m) \cdot P(m) = 2^{2^m} \cdot 2^{2^m} = 2^{2^{m+1}}$, where $A_j(m)$ is the number of matched paths through the level-m A-connection proto-WCFLOBDD to its jth exit vertex. However, we are just summing over all A-connection exit vertices, so $\sum_j A_j(m) = P(m)$.

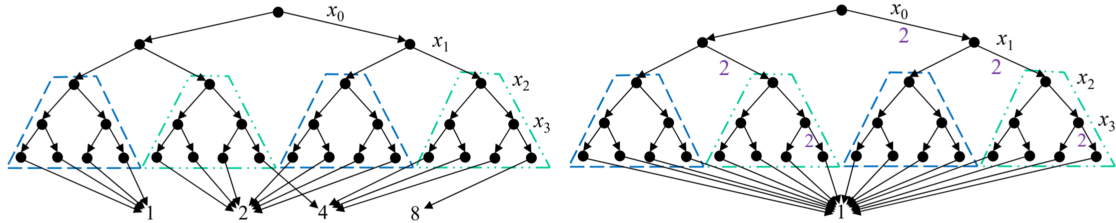
Obligation 2. This property is established by giving a (deterministic) construction to convert a decision tree with 2^{2^k} leaves into a level-k WCFLOBDD (satisfying all structural invariants). The construction is given in Appendix §A.1. (The construction is a recursive procedure that works greedily over the decision tree, from left to right.)

Obligation 3. This property is established by (i) unfolding WCFLOBDD C into a decision tree; (ii) folding the decision tree back to a WCFLOBDD C'; and (iii) showing that C' is isomorphic to C. (The folding-back step uses the same deterministic construction from Obligation 2.) □

The folding/unfolding constructions of Obligations 2 and 3 are used solely for the purpose of proving Thm. 4.1; they are not explicit operations on WCFLOBDDs.

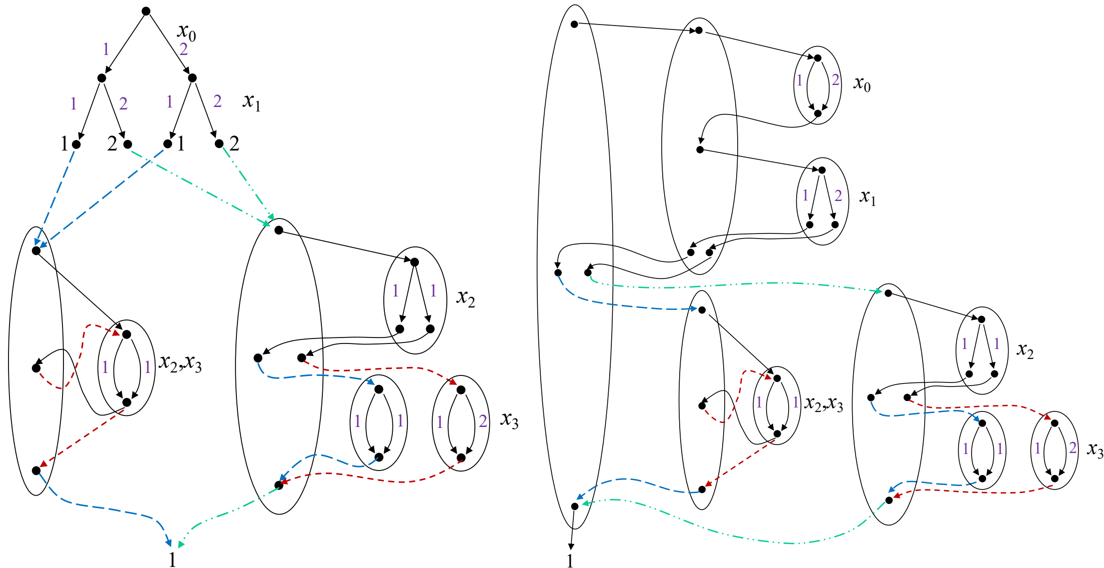
Intuition. Structural invariants (1)-(4), which deal with return tuples, ensure that the WCFLOBDD for a Boolean function f has a *"left-to-right"* folding of the decision tree for f (with the same variable ordering). Structural invariants (5)-(8), which deal with weights, ensure that there cannot be two proto-WCFLOBDDs for the same decision-tree structure that have different sets of level-0 groupings, i.e., with different (lw, rw) weights. The deterministic construction algorithm that folds a decision tree into a WCFLOBDD involves a two-step process. The decision tree is first converted into a weighted decision tree (a decision tree with weights on the edges), and then the weighted decision tree is folded into a WCFLOBDD. The first step enforces the structural invariants of the weights and the second step of *"left-to-right"* folding handles the structural invariants that deal with the return tuples.

Example: Fig. 4.3 shows the representation of a vector of length 16, and thus 4 variables are used to represent the index of an element of the vector. The four diagrams in Fig. 4.3 show some of the stages of converting a decision tree to a WCFLOBDD. Fig. 4.3(a) shows the decision-tree representation of the vector. The weighted decision tree for Fig. 4.3(a) is shown in Fig. 4.3(b). In this step, the weights are assigned to the edges, ensuring that the structural invariants on the weights hold. Fig. 4.3(c) shows the process of converting a weighted decision tree to a WCFLOBDD. The vertices labeled 1,2,1,2 at "half-height" (at the end of the weighted decision tree for x_0 and x_1) indicate the common sub-structures found during the left-to-right folding of the bottom half of the weighted decision tree. These common "values" are used to drive the left-to-right folding of the upper-half weighted decision tree, and to ensure that the structural invariants hold on the return tuples of the A-connection of the outermost grouping in Fig. 4.3(d). Finally, Fig. 4.3(d) shows the full WCFLOBDD representation for Fig. 4.3(a).



(a) Decision tree

(b) Weighted decision tree for (a). The edges labelled with numbers in purple are the edge weights. To reduce clutter, labels for edges with weight 1 are omitted.



(c) Hybrid of weighted decision tree for x_0 and x_1, and WCFLOBDDs for x_2 and x_3. The dashed, and dashed-double-dotted edges from the four vertices labeled 1, 2, 1, and 2, respectively, correspond to the dashed, and dashed-double-dotted trapezoids in (a) and (b). The numbers highlighted in purple indicate the edge-weights.

(d) WCFLOBDD representation for the weighted decision tree in (a) (Some of the groupings have been duplicated to avoid clutter). The numbers highlighted in purple indicate the edge-weights.

Figure 4.3: Representations of the vector $V = [1, 1, 1, 1, 2, 2, 2, 4, 2, 2, 2, 2, 4, 4, 4, 8]^T$ with variables $\langle x_0, x_1, x_2, x_3 \rangle$ representing an index into V.

Comparison with CFLOBDDs. WCFLOBDDs draw inspiration from CFLOBDDs: the structural elements of levels and A-connection/B-connection edges, along with structural invariants that enforce a left-to-right folding of the decision tree are similar to CFLOBDDs. However, in WCFLOBDDs the edges in level-0 groupings have weights, which differs from CFLOBDDs. To ensure that WCFLOBDDs are canonical, we introduced structural invariants (5)-(8) and (10).

4.2.3 Exponential Succinctness Gaps

In this section, we establish exponential gaps between (i) WCFLOBDDs and WBDDs, and (ii) WCFLOBDDs and CFLOBDDs using an example function.

Definition 4.6. The n-bit population-count function POP_n: $\{0,1\}^n \rightarrow \{2^j \mid j \in \{0 \dots n\}\}$ on variables $\{x_0 \cdots x_{n-1}\}$ is $POP_n(X) \overset{\text{def}}{=} 2^{pop(X)} = 2^{\sum_{i=0}^{n-1} \delta(x_i)}$, where $\delta(a) = \begin{cases} 1 & \text{a} = 1 \\ 0 & \text{otherwise} \end{cases}$ and $pop(X)$ represents the number of 1s in the binary representation of X.

Theorem 4.2. For $n = 2^k$, where $k \geq 0$, POP_n, defined over the variables $\{x_0 \cdots x_{n-1}\}$, can be represented by a WCFLOBDD with $\Theta(\log n)$ vertices and edges. A CFLOBDD that uses only the variables $\{x_0 \cdots x_{n-1}\}$ for POP_n requires $\Omega(n)$ vertices and edges; and a WBDD that uses the same variable set requires $\Theta(n)$ nodes.

Proof: Weighted Decision Tree. Each x_i node in a weighted decision tree for POP_n has a 1 on the left branch ($x_i = 0$) and a 2 on the right branch ($x_i = 1$). The paths of the weighted decision tree evaluate to $2^k + 1$ different values: the leftmost path (all variables set to 0) evaluates to $1 = 2^0$; the rightmost path (all variables set to 1) evaluates to 2^{2^k}. There are repeated values that the function represented by the weighted decision tree evaluates to, but we have every power of 2 from 2^0 to 2^{2^k}.

WBDD Claim. A WBDD for POP_n would

consist of 2^k don't-care nodes (one for every

variable)—with left-edge weight of 1 and right-edge weight of 2—stacked on top of each other. Hence, the size of the WBDD is linear in the number of variables: $\Theta(2^k) = \Theta(n)$. (See Fig. 4.4(i).)

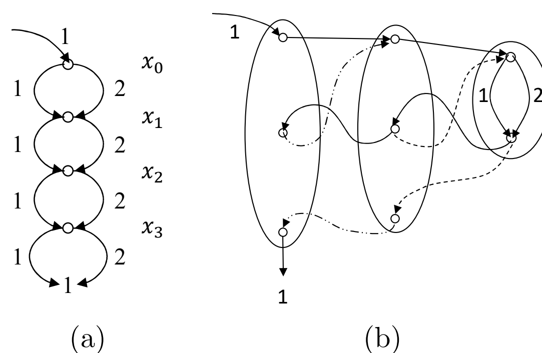


Figure 4.4: Two representations of POP_4 using (a) WBDDs and (b) WCFLOBDDs.

WCFLOBDD Claim. Consider a WCFLOBDD for POP_n W with $n = 2^k$ variables and k levels. There is only a single grouping in W for each level. At level 0, the unique grouping is a don't-care grouping with $(lw, rw) = (1, 2)$. At each level $l \geq 1$, the level-l grouping's A-connection and (one) B-connection both "call" the unique grouping at level $l - 1$. Because every level of W has only a single grouping, the WCFLOBDD has $\Theta(k) = \Theta(\log n)$ vertices and edges. (See Fig. 4.4(ii).)

CFLOBDD Claim. The terminal values of the CFLOBDD C representing POP_n would be the unique output values of the function. The terminal values of C would contain all powers of 2 from 2^0 to 2^{2^k}; hence C has $2^k + 1$ terminal values and the size of C is $\Omega(2^k) = \Omega(n)$.

Moreover, weighted decision trees, WBDDs, CFLOBDDs, and WCFLOBDDs always have the structures described above *regardless of the variable order one chooses to use*. Hence, for POP_n, WCFLOBDDs are inherently exponentially more succinct than both WBDDs and CFLOBDDs. □

4.3 Operations on WCFLOBDDs

In this section, we discuss the algorithms for operations on WCFLOBDDs. Tab. 4.1 shows a table with the operations on WCFLOBDDs, along with their complexities, as well as the complexities of the corresponding WBDDs operations.

Operation	Type Signature	Description	Time Complexity		
			WCFLOBDD	WBDD	CFLOBDD
Equal	WCFLOBDD × WCFLOBDD → Boolean	Checks if two WCFLOBDDs are equal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
ConstantZero	Int (k) → WCFLOBDD	Creates a WCFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.F$	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(k)$
ConstantOne	Int (k) → WCFLOBDD	Creates a WCFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.T$	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(k)$
Projection	Int(k) × Int(i) → WCFLOBDD	Creates a WCFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.x_i$	$\mathcal{O}(k)$	$\mathcal{O}(2^k)$	$\mathcal{O}(k)$
ScalarMultiply	WCFLOBDD (c) × Value (v) → WCFLOBDD	Performs $c' = c * v$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\|c\| \times \|c'\|)$
Pointwise Multiplication (Hadamard product $\odot$)	WCFLOBDD (c_1) × WCFLOBDD (c_2) → WCFLOBDD	Performs $c' = c_1 \odot c_2$	$\mathcal{O}(\|c_1\| \times \|c_2\| \times \|c'\|)$	$\mathcal{O}(\|c_1\|_B \times \|c_2\|_B)$	$\mathcal{O}(\|c_1\| \times \|c_2\| \times \|c'\|)$
Pointwise Addition	WCFLOBDD (c_1) × WCFLOBDD (c_2) → WCFLOBDD	Performs $c' = c_1 + c_2$	$\mathcal{O}(2^{vars})$	$\mathcal{O}(2^{vars})$	$\mathcal{O}(\|c_1\| \times \|c_2\| \times \|c'\|)$
PathWeight	WCFLOBDD(c) → WCFLOBDD	Computes the weight of paths to every exit vertex of every grouping	$\mathcal{O}(\|c\|)$	$\mathcal{O}(\|c\|_B)$	$\mathcal{O}(\|c\|)$
Sampling	WCFLOBDD(c) → String	Samples a path from c	$\mathcal{O}(\max(vars, \|c\|))$	$\mathcal{O}(\max(vars, \|c\|_B))$	$\mathcal{O}(\max(vars, \|c\|))$
KroneckerProduct	WCFLOBDD(c_1) × WCFLOBDD(c_2) → WCFLOBDD	Performs $c' = c_1 \otimes c_2$	$\mathcal{O}(1)$	$\mathcal{O}(\|c_1\|_B)$	$\mathcal{O}\left(\left(\begin{array}{c}\|c_1\| + \|c_2\| \\ + c_1.\#\text{exits} \times c_2.\#\text{exits}\end{array}\right) \times \|c'\|\right)$
MatrixMultiply	WCFLOBDD(c_1) × WCFLOBDD(c_2) → WCFLOBDD	Performs $c' = c_1 \times c_2$ for matrices of size $N \times N$	$\mathcal{O}(N^3)$, plus the time for a final call to `Reduce`	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$, plus the time for a final call to `Reduce`

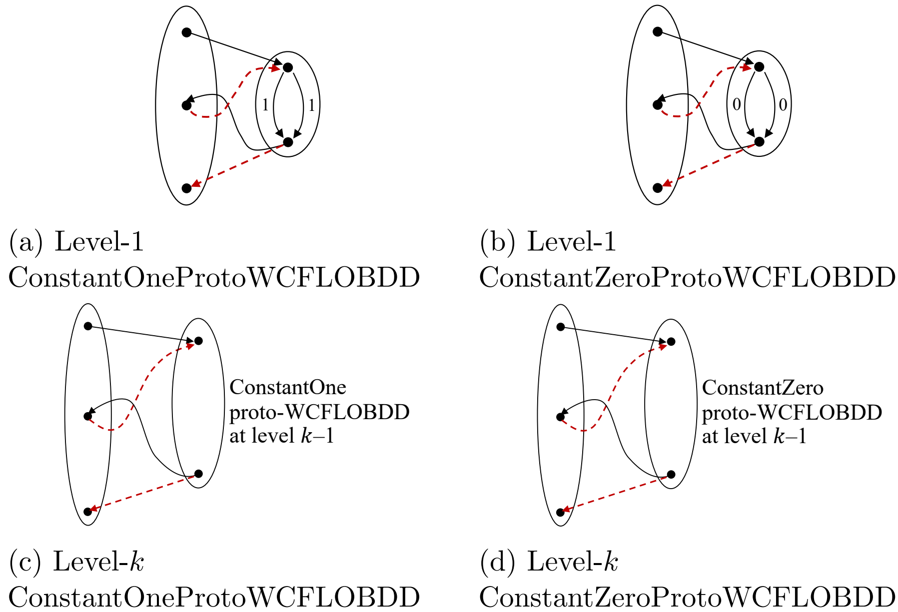
Table 4.1: List of operations on WCFLOBDDs, WBDDs and CFLOBDDs; *vars* denotes the number of Boolean variables ($= 2^k$, where k is the number of levels of the WCFLOBDD). The size measure $|\cdot|$ counts the number of vertices and edges—with no double-counting of vertices and edges in groupings that are shared due to hash-consing. In the column for the time complexities of WBDD operations, an occurrence of c refers to a WBDD argument of the operation, and $|c|_B$ denotes the size of WBDD c (the number of nodes and edges). Note that the complexity of MatrixMultiply is in terms of the sizes of the matrices represented by c_1 and c_2, and not the sizes of the data structures c_1 and c_2, plus the cost of an added "final call to `Reduce`." The latter cost depends on the sizes of the structures that are input to and output from `Reduce`. That is, the complexity of $c' = \texttt{Reduce}(c)$ is $\mathcal{O}(|c'| \times |c|)$. Because it is difficult to combine the complexity of the symbolic matrix-multiplication computation and that of `Reduce`, we broke out the cost of `Reduce` as a separate item.

Because WCFLOBDDs are hierarchically structured, algorithms are divide-and-conquer algorithms (similar to the algorithms in CFLOBDDs) that, for a given `InternalGrouping`, recurse on the A-connection and then the B-connections (or vice versa), thereby splitting the variables in half for each subproblem, along the following lines:

```
Op(Grouping g) {
  ... // Base case:  Construct appropriate level-0 grouping
  InternalGrouping g' = new InternalGrouping(k); // k = g.level
  // Recursive calls on Op(g.AConnection) and Op(g.BConnections[i]),
  // for 1 ≤ i ≤ numberOfBConnections, to fill in g'.AConnection and the
  // elements of array g'.BConnections with level-(k-1) Groupings
  ...
  return RepresentativeGrouping(g');
}
```

For each `InternalGrouping` constructed, the algorithms enforce the structural invariants (§4.2.2) on return tuples and weights (at level 0) so that the return value is a (proto-)WCFLOBDD. The algorithms use two standard techniques (mostly elided in the pseudo-code) to ensure that the amount of memory used to represent a WCFLOBDD does not blow up, and to reduce the cost of WCFLOBDD operations. Techniques such as (a) *Hash consing* Goto (1974), (b) Function Caching Michie (1967), used in §3.4, are also used in the algorithms for WCFLOBDDs. *Hash consing* Goto (1974) is used to ensure that only one representative of a value exists in memory. The operation `RepresentativeGrouping` checks whether a grouping is a duplicate, and if so, discards it and returns the grouping's representative. `RepresentativeGrouping` consults a global hash-table that maintains pointers to the unique representation of each grouping. Consequently, one can test in unit time if two proto-WCFLOBDDs are equal by comparing their pointers. (As will be seen in lines [2]–[5] of Alg. 31, this ability allows some special cases to be identified quickly and helps speed up computations.)

In discussions of space costs later in this section, the "size of a WCFLOBDD"



(a) Level-1 ConstantOneProtoWCFLOBDD

(b) Level-1 ConstantZeroProtoWCFLOBDD

(c) Level-k ConstantOneProtoWCFLOBDD

(d) Level-k ConstantZeroProtoWCFLOBDD

Figure 4.5: `ConstantOneProtoWCFLOBDD` and `ConstantZeroProtoWCFLOBDD` for levels 1 and $k > 1$.

is the total number of vertices and edges—with no double-counting of vertices and edges in groupings that are shared due to hash-consing.

A *cache* for a function F is an associative-lookup table with pairs of the form $[x, F(x)]$, used to eliminate the cost of re-doing a previously performed computation Michie (1967). All the algorithms on WCFLOBDDs perform function caching using the following idiom:

```
F(x) {
    if cache_F(x) ≠ NULL return cache_F(x);
    ...
    cache_F(x) = retVal; // Update the cache with the return value
    return retVal;
}
```

At the end of every algorithm, the cache is updated with the computed result, thereby allowing a redundant computation to be avoided if the algorithm is called later with the same arguments.

4.3.0.1 Constant Functions

Fig. 4.5 shows the proto-WCFLOBDDs for the constant functions $f_{\bar{0}}(x) = \lambda x.\bar{0}$ and $f_{\bar{1}}(x) = \lambda x.\bar{1}$ (`ConstantZero` and `ConstantOne`, respectively). These proto-WCFLOBDDs each have just one kind of level-0 grouping. Their value tuples are $[\bar{0}]$ and $[\bar{1}]$, respectively.

4.3.0.2 Projection Function

Create a level-k proto-WCFLOBDD g for the function $f(k,i) \stackrel{\text{def}}{=} \lambda x_0 \ldots x_{2^k-1}.x_i$. If $i < 2^{k-1}$, g's A-connection is a proto-WCFLOBDD for $f(k-1,i)$, and g has two middle vertices. If $i \neq 0$, then the 1st B-connection of g is `ConstantOne` and the 2nd B-connection is `ConstantZero`; if $i = 0$, the B-connections are the opposite. If $i \geq 2^{k-1}$, g's A-connection is `ConstantOne` and g has one middle vertex; g's only B-connection is a proto-WCFLOBDD for $f(k-1, i-2^{k-1})$. In all cases, g has two exit vertices; they lead to the value tuple $[\bar{1}, \bar{0}]$ in all cases except for $i = 0$, when they lead to $[\bar{0}, \bar{1}]$.

4.3.0.3 Unary Operations

Scalar Multiplication. Given a scalar $v \in \mathcal{D}$ and WCFLOBDD $C = \langle fw, g, vt\rangle$ for function f, the WCFLOBDD for $\lambda x.(v \cdot f(x))$ is $C' = \begin{cases} \langle v \cdot fw, g, vt\rangle & v \neq \bar{0} \\ \texttt{ConstantZero} & v = \bar{0} \end{cases}$

4.3.0.4 Pointwise Binary Operations

A binary operation *op* works *pointwise* if, for two functions f and g, $f \, op \, g \stackrel{\text{def}}{=} \lambda x.f(x) \, op \, g(x)$. We discuss $op \in \{\cdot, +\}$. The algorithms operate on WCFLOBDDs that are parametrized on the semi-field $\mathcal{D}$, and hence, the operations $\cdot$ and $+$ are specific to $\mathcal{D}$.

Pointwise Multiplication (Alg. 30). Given the WCFLOBDDs for functions f and g, the goal is to create the WCFLOBDD for their pointwise product, $f \cdot g$. Let

Algorithm 30: Pointwise Multiplication

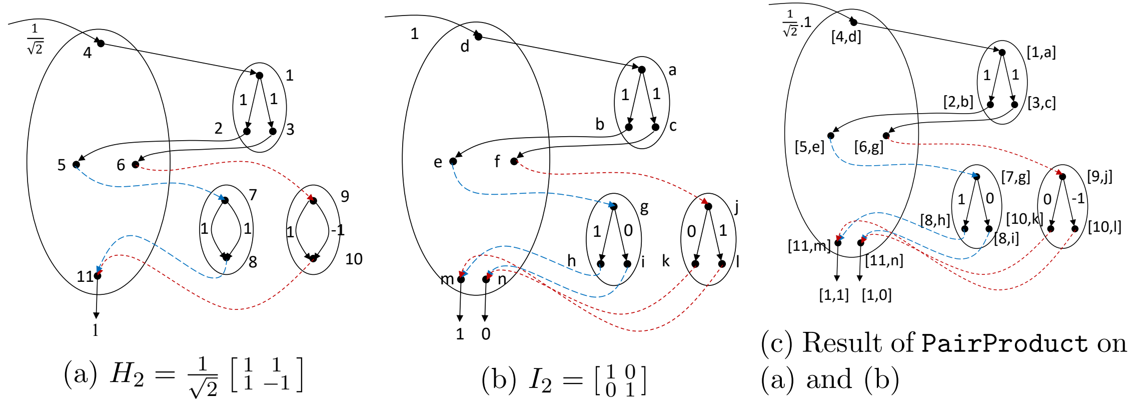
Input: WCFLOBDDs n1 = $\langle fw1, g1, vt1\rangle$, n2 = $\langle fw2, g2, vt2\rangle$
Output: WCFLOBDD n = n1 · n2

```
1  begin
       // Perform cross product
2      Grouping×PairTuple [g,pt] = PairProduct(g1,g2);
3      ValueTuple deducedValueTuple = [ vt1[i1] · vt2[i2] : [i1,i2] ∈ pt ];
       // Collapse duplicate leaf values, folding to the left
4      Tuple×Tuple [inducedValueTuple,inducedReductionTuple] =
         CollapseClassesLeftmost(deducedValueTuple);
5      Grouping×Weight [g', fw] = Reduce(g, inducedReductionTuple,
         deducedValueTuple);
6      WCFLOBDD n = RepresentativeCFLOBDD(fw · fw1 · fw2, g',
         inducedValueTuple);
7      return n;
8  end
```

$c_1 = \langle fw_1, g_1, v_1\rangle$ and $c_2 = \langle fw_2, g_2, v_2\rangle$ be the WCFLOBDDs that represent f and g, respectively. As with BDDs, such operations on WCFLOBDDs can be implemented via a two-step process: (i) create a cross-product of c_1 and c_2, and (ii) perform a reduction step on the result of step (i). The cross-product (called PairProduct) is performed recursively on the A-connection groupings, followed by the B-connection groupings (see Fig. 4.6 and Algs. 31 and 32). The cross-product of two groupings yields a tuple of the form (g, [pt]), where g is the resultant grouping and pt is a sequence of index-pairs. The index-pairs in pt indicate the B-connection groupings on which cross-product operations need to be performed. Fig. 4.6(c) shows the cross-product of the WCFLOBDDs that represent $H_2 = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right]$ and $I_2 = \left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right]$.

At level-0 and the topmost level, the PairProduct algorithm performs the following steps:

- Level-0: Let the two level-0 groupings be g_1^0 with weights (lw_1, rw_1) and g_2^0 with weights (lw_2, rw_2). The return value is a new level-0 grouping g, with weights $(lw_1 \cdot lw_2, rw_1 \cdot rw_2)$, along with a sequence of index-pairs on which the cross-product is performed next. For example, if g_1^0 is a fork-grouping and g_2^0 is a



(a) $H_2 = \frac{1}{\sqrt{2}} \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$

(b) $I_2 = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$

(c) Result of `PairProduct` on (a) and (b)

Figure 4.6: Illustration of `PairProduct` of the WCFLOBDDs for matrices H_2 and I_2.

don't-care grouping, then $pt = [(1,1),(2,1)]$.

- Topmost level: If (g, pt) is returned after performing the cross-product, the indices in pt are indices into the value tuples of c_1 and c_2. A new value tuple v is constructed accordingly. For instance, if $v_1 = [\bar{1}, \bar{0}]$, $v_2 = [\bar{0}, \bar{1}]$, and $pt = [(1,1),(1,2),(2,1)]$, then v is $[(v_1[1] \cdot v_2[1]), (v_1[1] \cdot v_2[2]), (v_1[2] \cdot v_2[1])] = [\bar{1} \cdot \bar{0}, \bar{1} \cdot \bar{1}, \bar{0} \cdot \bar{0}] = [\bar{0}, \bar{1}, \bar{0}]$.

When the resulting value tuple has duplicate entries, such as $v = [\bar{0}, \bar{1}, \bar{0}]$ in the example above, it is necessary to perform a reduction step, `Reduce` (Algs. 63 and 64 in §L), to maintain the WCFLOBDD structural invariants. In this case, `Reduce` folds together the first and third exit vertices of g, which are both mapped to $\bar{0}$ by v. In Alg. 30, the information that directs the reduction step is obtained by calling CollapseClassesLeftmost (Alg. 33), which returns two tuples: inducedValueTuple (here, $[\bar{0}, \bar{1}]$) and inducedReductionTuple (here, $[1, 2, 1]$): inducedValueTuple consists of the leftmost occurrences of $\bar{0}$ and $\bar{1}$ in v, in the same left-to-right order in which they occur in v; inducedReductionTuple indicates where each occurrence of $\bar{0}$ and $\bar{1}$ in v is mapped to in inducedValueTuple. `Reduce` traverses g backwards to create g', a reduced version of g that, together with inducedValueTuple as its terminal values, satisfies the WCFLOBDD structural invariants. The result from $c_1 \cdot c_2$ is the

```
Algorithm 31: PairProduct
   Input: Groupings g1, g2
   Output: Grouping g: product of g1 and g2; PairTuple ptAns: tuple of
           pairs of exit vertices
 1 Begin
 2 |  if g1 and g2 are both ConstantOneProtoCFLOBDD then return [
 |      g1, [[1,1]] ];
 3 |  if g1 or g2 is ConstantZeroProtoCFLOBDD then return [
 |      ConstantZeroProtoCFLOBDD(g1.level), [[1,1]] ];
 4 |  if g1 is ConstantOneProtoCFLOBDD then return [ g2, [[1,k] : k ∈
 |      [1..g2.numberOfExits]] ];
 5 |  if g2 is ConstantOneProtoCFLOBDD then return [ g1, [[k,1] : k ∈
 |      [1..g1.numberOfExits]] ];
   |  // Elided: similar cases for other base cases, with
   |     appropriate pairings of exit vertices
 6 |  if g1 and g2 are fork groupings then
 7 |  |  ForkGrouping g = new ForkGrouping(g1.lw · g2.lw, g1.rw · g2.rw);
 8 |  |  return [ g, [[1,1],[2,2]] ];
 9 |  end
   |  // Pair the A-connections
10 |  Grouping×PairTuple [gA,ptA] = PairProduct(g1.AConnection,
 |      g2.AConnection);
11 |  InternalGrouping g = new InternalGrouping(g1.level);
12 |  g.AConnection = gA;
13 |  g.AReturnTuple = [1..|ptA|]; // Represents the middle vertices
14 |  g.numberOfBConnections = |ptA|;
```

WCFLOBDD $\langle fw_1 \cdot fw_2, g', \text{inducedValueTuple}\rangle$.

Just as there can be multiple occurrences of a given node in a BDD, there can be multiple occurrences of a given grouping in a WCFLOBDD. To avoid a blow-up in costs, binary operations need to avoid making repeated calls on a given pair of groupings $h_1 \in c_1$ and $h_2 \in c_2$. Assuming that the hashing methods used for hash-consing and function caching run in expected unit-cost time, the cost of PairProduct is bounded by the product of the sizes of the two-argument WCFLOBDDs, and the cost of `Reduce` is bounded by the product of the sizes of its input and output WCFLOBDDs.

Algorithm 32: PairProduct (cont.)

```
    Input: Groupings g1, g2
    Output: Grouping g: product of g1 and g2; PairTuple ptAns: tuple of
            pairs of exit vertices
15
      // Pair the B-connections, but only for pairs in ptA
      // Descriptor of pairings of exit vertices
16    Tuple ptAns = [];
      // Create a B-connection for each middle vertex
17    for j ← 1 to |ptA| do
18        Grouping×PairTuple [gB,ptB] =
            PairProduct(g1.BConnections[ptA(j)(1)],
            g2.BConnections[ptA(j)(2)]);
19        g.BConnections[j] = gB ;
          // Create g.BReturnTuples[j], and augment ptAns as
             necessary
20        g.BReturnTuples[j] = [] ;
21        for i ← 1 to |ptB| do
22            c1 = g1.BReturnTuples[ptA(j)(1)](ptB(i)(1));    // a g1 exit
23            c2 = g2.BReturnTuples[ptA(j)(2)](ptB(i)(2)) ;   // a g2 exit
24            if [c1,c2] ∈ ptAns then     // Not a new exit vertex of g
25                index = the k such that ptAns(k) == [c1,c2] ;
26                g.BReturnTuples[j] = g.BReturnTuples[j] || index ;
27            else                 // Identified a new exit vertex of g
28                g.numberOfExits = g.numberOfExits + 1 ;
29                g.BReturnTuples[j] = g.BReturnTuples[j] ||
                    g.numberOfExits ;
30                ptAns = ptAns || [c1,c2] ;
31            end
32        end
33    end
34    return [RepresentativeGrouping(g), ptAns];
35 end
```

Pointwise Addition (Algs. 66, 67, and 68 in §M.2). Given the WCFLOBDDs $f = \langle fw_1, h_1, v_1\rangle$ and $g = \langle fw_2, h_2, v_2\rangle$, the goal is to create the WCFLOBDD for their pointwise sum, $f + g$. The algorithm performs a kind of "weighted" cross-product of f and g. The algorithm is similar to the one for pointwise multiplication, but with a

Algorithm 33: CollapseClassesLeftmost

```
  Input: Tuple equivClasses
  Output: Tuple×Tuple [projectedClasses, renumberedClasses]
1 begin
    // Project the tuple equivClasses, preserving
       left-to-right order, retaining the leftmost instance of
       each class
2   Tuple projectedClasses = [equivClasses(i) : i ∈ [1..|equivClasses|] | i =
      min{j ∈ [1..|equivClasses|] | equivClasses(j) = equivClasses(i)}];
    // Create tuple in which classes in equivClasses are
       renumbered according to their ordinal position in
       projectedClasses
3   Map orderOfProjectedClasses = {[x,i]: i ∈ [1..|projectedClasses|] | x =
      projectedClasses(i)};
4   Tuple renumberedClasses = [orderOfProjectedClasses(v) : v ∈
      equivClasses];
5   return [projectedClasses, renumberedClasses];
6 end
```

few embellishments—in particular, weights are aggregated at each level and used in the recursive computation at the next level. The weighted cross-product construction is performed recursively on the A-connections, followed by additional recursive calls on the B-connections according to sets of B-connection indices included in the answer returned from the call on the A-connections. In addition to two groupings, g_1 and g_2, the cross-product also takes as input two weights, p_1 and p_2. To get things started, the top-level call is WeightedPairProduct(h_1,h_2, fw_1, fw_2).

The return value takes the form $[g, pt]$, where g is a grouping that is temporary (and later will be updated with actual weights) and pt is a sequence of tuples of the form $\langle(q_1, i_1), (q_2, i_2)\rangle$. Values such as i_1 and i_2 are exit-vertex indices—used to determine what further recursive calls to invoke—and q_1 and q_2 are weights that play the role of p_1 and p_2 in those recursive calls—i.e., WeightedPairProduct(_, _, q_1, q_2), where the groupings used as the first and second arguments are based on i_1 and i_2, respectively.

Level 0: Base-case processing happens at level 0. Let the two level-0 groupings be g_1^0 with weights (lw_1, rw_1) and g_2^0 with weights (lw_2, rw_2). Suppose that the input weights are p_1 and p_2. A new level-0 grouping g is created with weights (x_1, x_2), where $x_1 = \bar{1}$ in all cases except when $p_1 \cdot lw_1 = \bar{0}$, in which case $x_1 = \bar{0}$, and $x_2 = \bar{1}$ in all cases except when $p_2 \cdot lw_2 = \bar{0}$, in which case $x_2 = \bar{0}$. The return values are g and pt, where pt contains two tuples indicating which cross-products are to be performed subsequently.

The following table shows how pt is constructed:

g_1^0	g_2^0	pt
`DontCareGrouping`	`DontCareGrouping`	$[\langle(p_1 \cdot lw_1, 1), (p_2 \cdot lw_2, 1)\rangle,$ $\langle(p_1 \cdot rw_1, 1), (p_2 \cdot rw_2, 1)\rangle]$
`ForkGrouping`	`DontCareGrouping`	$[\langle(p_1 \cdot lw_1, 1), (p_2 \cdot lw_2, 1)\rangle,$ $\langle(p_1 \cdot rw_1, 2), (p_2 \cdot rw_2, 1)\rangle]$
`DontCareGrouping`	`ForkGrouping`	$[\langle(p_1 \cdot lw_1, 1), (p_2 \cdot lw_2, 1)\rangle,$ $\langle(p_1 \cdot rw_1, 1), (p_2 \cdot rw_2, 2)\rangle]$
`ForkGrouping`	`ForkGrouping`	$[\langle(p_1 \cdot lw_1, 1), (p_2 \cdot lw_2, 1)\rangle,$ $\langle(p_1 \cdot rw_1, 2), (p_2 \cdot rw_2, 2)\rangle]$

Topmost level: Suppose that (fg, pt) is returned by the top-level call on WeightedPairProduct. Some post-processing takes place to set up an appropriate call on `Reduce`. In each tuple $\langle(q_1, i_1), (q_2, i_2)\rangle$ in pt, i_1 and i_2 are indices into the value tuples v_1 and v_2, respectively, of f and g. For instance, if $v_1 = [\bar{1}, \bar{0}]$, $v_2 = [\bar{1}]$, and $pt = [\langle(d_1, 1), (d_2, 1)\rangle, \langle(d_3, 2), (d_4, 1)\rangle]$, where $d_1, d_2, d_3, d_4 \in \mathcal{D}$, then a tuple v is created: $v = [(d_1 \cdot v_1[1] + d_2 \cdot v_2[1]), (d_3 \cdot v_1[2] + d_4 \cdot v_2[1])] = [d_1 + d_2, d_4]$. fg is then reduced with respect to v, i.e., v is propagated through fg to create fg' so that (i) the edges of level-0 groupings in fg' hold appropriate weights, and (ii) fg' satisfies the structural invariants of Defn. 4.5. The reduction operation also returns the new factor weight fw'. For our example, assuming that neither $d_1 + d_2$ nor d_4 is $\bar{0}$, the overall top-level tuple of terminal values would be $v' = [\bar{1}]$: v' is obtained from v by replacing every non-zero element in v with $\bar{1}$—to obtain $v'' = [\bar{1}, \bar{1}]$—and then v' retains only the first occurrences of the different values in v''. The resultant WCFLOBDD is $h = f + g = \langle fw', fg', v'\rangle$.

The pseudo-code is given as Algs. 66, 67, and 68 in §M.2. Note that several optimizations can be performed for special cases, such as `ConstantZero` or cases when both groupings are equal, to avoid performing a traversal of all the levels of groupings.

4.3.0.5 Representing Matrices and Vectors

When matrices are represented with WCFLOBDDs, the variables correspond to the bits of the matrix's row and column indices. Similar to CFLOBDDs, for a matrix M of size $2^n \times 2^n$, the WCFLOBDD representation has $2n$ variables $(x_0, \ldots, x_{n-1})$ and $(y_0, \ldots, y_{n-1})$, where the x-variables are the row-index bits and the y-variables are the column-index bits. Typically, we use a variable ordering in which the x and y variables are interleaved, x_0 and y_0 are the most-significant bits, and x_{n-1} and y_{n-1} are the least-significant bits. The nice property of this ordering is that, as we work through each pair of variables in an assignment, the matrix elements that remain "in play" represent a sub-block of M.

When vectors of size $2^n \times 1$ are represented using WCFLOBDDs, the variables $(x_0, \ldots, x_{n-1})$ correspond to the bits of the vector's row index. Typically, x_0 is the most-significant bit and x_{n-1} is the least-significant bit.

4.3.0.6 Kronecker Product

Consider two matrices M_1 and M_2 represented by WCFLOBDDs $c_1 = \langle fw_1, g_1, v_1 \rangle$ and $c_2 = \langle fw_2, g_2, v_2 \rangle$. Their Kronecker product $M = M_1 \otimes M_2$ is performed by using the "procedure-call" mechanism of WCFLOBDDs to "stack" the Boolean variables of c_1 on top of those of c_2. If c_1 or c_2 is `ConstantZero`, then g = `ConstantZero`. Otherwise, a new grouping g is created with g_1 as the A-connection of g, and g_2 as the B-connection(s) of g. If there exists an exit vertex e of g_1 that has a terminal value of $\bar{0}$, then the middle vertex of g that is connected to e has a B-connection to a `ConstantZeroProtoWCFLOBDD`. The terminal values of g

are as follows:

$$v = \begin{cases} v_2 & \text{if } v_1 == [\bar{1}] \text{ (i.e., no such } e \text{ exists)} \\ [\bar{0}, \bar{1}] & \text{if } e \text{ is the } 1^{st} \text{ exit vertex of } g_1 \\ [\bar{1}, \bar{0}] & \text{if } e \text{ is the } 2^{nd} \text{ exit vertex of } g_1 \text{ and } v_2 = [\bar{1}] \\ v_2 & \text{otherwise} \end{cases}$$

The resultant WCFLOBDD $c = c_1 \otimes c_2$ is $\langle fw_1 \cdot fw_2, g, v \rangle$.

4.3.0.7 Matrix Multiplication

Matrix multiplication is performed by a recursive divide-and-conquer algorithm, where the divide step performs a block decomposition that reduces the problem size to $\sqrt{N} \times \sqrt{N}$, and the (recursive) conquer step is similar to the standard cubic-time algorithm. As discussed in §3.6.7, WCFLOBDDs also follow a divide-and-conquer algorithm, first solving subproblems on A-connections and then B-connections.[1] In essence, one splits on half of the Boolean variables, which leads to $O(\sqrt{N})$ problems, each of size $O(\sqrt{N})$ Fig. 3.10.

Consider two $N \times N$ matrices P and P', represented by WCFLOBDDs $C = \langle fw, g, v \rangle$ and $C' = \langle fw', g', v' \rangle$ using the interleaved-variable order. The A-connections of g and g' represent the commonalities in sub-blocks of P and P', respectively, of size $\sqrt{N} \times \sqrt{N}$ and the B-connections of g and g' represent the sub-matrices of P and P', respectively, of size $\sqrt{N} \times \sqrt{N}$. The matrix-multiplication algorithm is recursively called on the A-connections of g and g', followed by the B-connections, based on information returned by the call on the A-connections, and then possibly some matrix-addition operations.

The challenge that we face is that at all levels below top-level, various sub-matrices of the left argument need to be multiplied by various sub-matrices of the right argument, and added together. However, for the computation performed at a

[1]Some other divide-and conquer algorithms on WCFLOBDDs solve subproblems on B-connections and then A-connections.

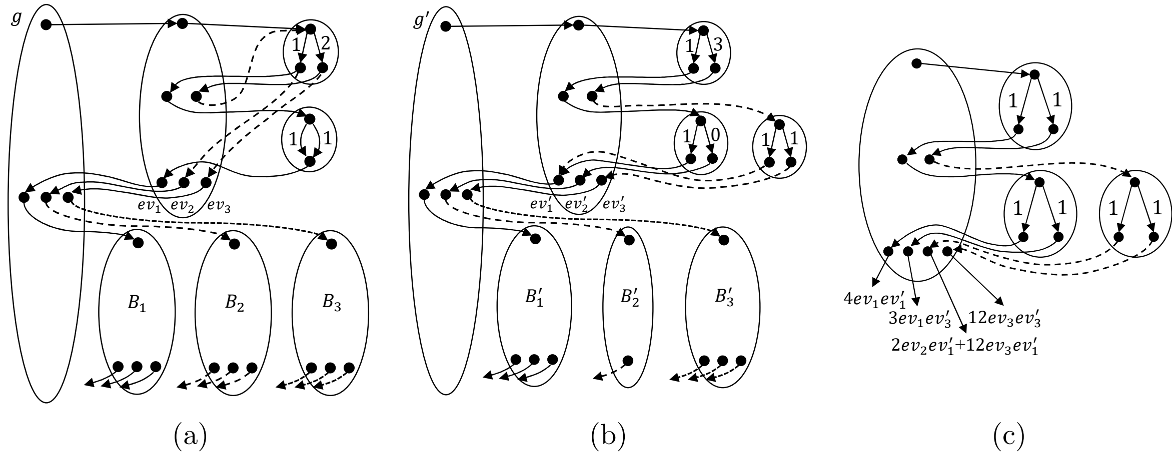


Figure 4.7: Illustration of how bilinear polynomials over exit vertices of lower-level groupings arise in matrix multiplication. (a) Left argument of $g \times g'$; (b) right argument of $g \times g'$; (c) the level-1 structure that is constructed in the level-1 subproblem g.AConnection $\times$ g'.AConnection.

given level of the WCFLOBDD, the algorithm has neither values nor sub-matrices at hand. Those (unknown) sub-matrices correspond to the exit vertices of the groupings that the current invocation of the algorithm was passed as the left and right arguments. Call these sets of exit vertices EV and EV', respectively. We can use the elements of EV and EV' as variables, and compute non-top-level matrix multiplications symbolically. It turns out that the symbolic information we need to keep takes the form of bilinear polynomials over EV and EV', consisting of summands of the form $c \cdot ev_i \cdot ev'_j$, where $ev_i \in EV$ and $ev'_j \in EV'$.

Example 4.1. Fig. 4.7 illustrates of how such bilinear polynomials over exit vertices arise in matrix multiplication. Fig. 4.7(a) and (b) show level-2 groupings, g and g', which are the left-hand and right-hand arguments of a matrix-multiplication (sub)problem. The first step of this multiplication problem is to symbolically multiply the level-1 groupings g.AConnection and g'.AConnection. In Fig. 4.7(a) and (b), when considered as 2×2 matrices over their respective exit vertices, $[ev_1, ev_2, ev_3]$ and $[ev'_1, ev'_2, ev'_3]$, g.AConnection and g'.AConnection are the matrices of bilinear

functions shown on the left side of Eqn. (4.1):

$$\begin{bmatrix} ev_1 & ev_1 \\ 2ev_2 & 4ev_3 \end{bmatrix} \times \begin{bmatrix} ev'_1 & 0ev'_2 \\ 3ev'_1 & 3ev'_3 \end{bmatrix} = \begin{bmatrix} 4ev_1 ev'_1 & 3ev_1 ev'_3 \\ 2ev_2 ev'_1 + 12ev_3 ev'_1 & 12ev_3 ev'_3 \end{bmatrix} \tag{4.1}$$

Fig. 4.7(c) shows the level-1 preliminary structure that is constructed after multiplying g.AConnection and g'.AConnection. This structure represents the matrix on the right-hand side of Eqn. (4.1). Each exit vertex of Fig. 4.7(c) is associated with a bilinear polynomial consisting of summands of the form $c \cdot ev_i \cdot ev'_j$, where $ev_i \in EV$ and $ev'_j \in EV'$. (In essence, the structure is a CFLOBDD with a bilinear polynomial for each terminal value.) The interpretation of the bilinear monomials from Fig. 4.7(c) leads to five B-connection multiplications being performed: $B_1 \times B'_1$, $B_1 \times B'_3$, $B_2 \times B'_1$, $B_3 \times B'_1$, and $B_3 \times B'_3$. Each multiplication produces a structure that is similar to the one shown in Fig. 4.7(c). Five scalar multiplications and one matrix addition are then performed. These last steps again produce bilinear polynomials because bilinear polynomials are closed under linear arithmetic (see Eqn. (4.2)).

The matrix-multiplication algorithm works recursively level-by-level, creating WCFLOBDDs for which the entries in the value tuples are bilinear polynomials. We call such value tuples *MatMultTuple*s. A *MatMultTuple* is a sequence of bilinear polynomials over the exit vertices of two groupings g_1 and g_2. Each bilinear polynomial bp is a map from a pair of exit-vertex indices to a coefficient (which is a value in $\mathcal{D}$): $bp \in BP_{EV,EV'} \overset{\text{def}}{=} (EV \times EV') \to \mathcal{D}$, where EV, EV' are the sets of exit vertices of g_1 and g_2. To perform linear arithmetic on bilinear polynomials, we define

$$\begin{array}{lr} 0_{BP} : BP & 0_{BP} \overset{\text{def}}{=} \lambda(ev, ev') \,.\, \bar{0} \\ + : BP \times BP \to BP & bp_1 + bp_2 \overset{\text{def}}{=} \lambda(ev, ev') \,.\, bp_1(ev, ev') + bp_2(ev, ev') \\ \cdot : \mathcal{D} \times BP \to BP & n \cdot bp \overset{\text{def}}{=} \lambda(ev, ev') \,.\, n \cdot bp(ev, ev') \end{array} \tag{4.2}$$

The base case is for two level-1 proto-WCFLOBDDs, which correspond to a pair of 2×2 matrices. The left and right decision-edges of a level-1 proto-WCFLOBDD's level-0 groupings hold weights, which serve as coefficients of *linear* functions in the entries of each of the 2×2 matrices. The result of multiplying two

level-1 proto-WCFLOBDDs introduces bilinear polynomials.[2] Operationally, a bilinear polynomial can be represented by a set of triples of the form $\{\ldots, [(i,j), c], \ldots\}$, where $[(i,j), c]$ represents a term of the form $c \cdot ev_i \cdot ev'_j$.

Example 4.2. Returning to Ex. 4.1, each entry in the matrix on the right-hand side of Eqn. (4.1) can be represented by a set of triples.

$$\begin{bmatrix} \{[(1,1),4]\} & \{[(1,3),3]\} \\ \{[(2,1),2],[(3,1),12]\} & \{[(3,3),12]\} \end{bmatrix}$$

The *MatMultTuple* is the listing of exit vertices for interleaved-variable order:

$$[\{[(1,1),4]\}, \{[(1,3),3]\}, \{[(2,1),2],[(3,1),12]\}, \{[(3,3),12]\}].$$

More abstractly, the function bp holds the coefficients of the bilinear polynomial over variable pairs $\{ev \in EV\} \times \{ev' \in EV'\}$, and bp implicitly denotes the expression

$$\sum_{ev \in EV \times ev \in EV'} bp(ev, ev') \cdot ev \cdot ev'.$$

As symbolic matrix multiplication is performed, a key operation is to "evaluate" a bilinear polynomial with respect to a binding of exit-vertices to (other) bilinear polynomials. So why don't we get quartic polynomials, and then even higher-degree polynomials? Because all operations are performed on bilinear polynomials over the same variable sets (i.e., sets of exit vertices). Consider again Fig. 4.7(c) and the bilinear polynomials associated with each exit vertex. Each such polynomial can be considered to be a bilinear polynomial over the middle vertices of g and g', and thus consists of terms of the form $c \cdot m_i \cdot m'_j$. Each such term is treated as a directive to multiply g.BConnections$[i]$ and g'.BConnections$[j]$.

For instance, the third exit vertex of Fig. 4.7(c) has the associated bilinear polynomial $pA = 2ev_2ev'_1 + 12ev_3ev'_1$. The first term, $2ev_2ev'_1$, is evaluated with respect to the binding $[ev_2 \mapsto B_2, ev'_1 \mapsto B'_1]$, leading to $B_2 \times B'_1$; the second term,

[2]As explained shortly, more complicated polynomials do not arise at levels $2, 3, \ldots$

$12 ev_3 ev_1'$, is evaluated with respect to the binding $[ev_3 \mapsto B_3, ev_1' \mapsto B_1']$, leading to $B_3 \times B_1'$. A multiplication of g.BConnections$[i]$ and g'.BConnections$[j]$ produces a matrix $m^{i,j}$ whose $(k,l)^{th}$ entry is a bilinear polynomial $pB_{k,l}^{i,j}$ over the exit vertices of g.BConnections$[i]$ and g'.BConnections$[j]$. Now—and this observation is the key reason why everything stays bilinear—g.BReturnTuples$[i]$ and g'.BReturnTuples$[j]$ are used to convert $pB_{k,l}^{i,j}$ into a bilinear polynomial $p_{k,l}^{i,j}$ over the *exit vertices of* g *and* g'. Consequently, the remaining steps of "evaluating" pA—multiplying a bilinear polynomial such as $p_{k,l}^{i,j}$ by a constant and addition of bilinear polynomials—are all performed on bilinear polynomials with the same variable sets, namely, the exit vertices of g and g'. Bilinear polynomials are closed under these operations (Eqn. (4.2)).

At top level, we form a preliminary value tuple w by evaluating each bilinear polynomial bp in the top-level *MatMultTuple* as follows:

$$\langle v, v' \rangle (bp) \stackrel{\text{def}}{=} \sum \{ bp(ev, ev') \cdot v(ev) \cdot v'(ev') \mid ev \in EV, ev' \in EV' \}$$

By structural invariant (10), the value tuple $\tilde{w}$ of the answer WCFLOBDD must be one of $\{[\bar{0}], [\bar{1}], [\bar{0}, \bar{1}], [\bar{1}, \bar{0}]\}$. Thus, $\tilde{w}$ is constructed from w by replacing all non-$\bar{0}$ entries of w with $\bar{1}$ and removing duplicates. To ensure that the final answer satisfies the WCFLOBDD structural invariants, `Reduce` is called (Algs. 63 and 64 in §L), which is passed the pattern of repeated values in w (along with w itself).

Certain optimizations can be performed to avoid traversing all the levels of the argument groupings: when at least one argument is `ConstantZero`, then all paths have weight $\bar{0}$, and the computation can be short-circuited. Similarly, when one of the arguments is the identity matrix, we can return the other argument.

4.3.0.8 Sampling

A WCFLOBDD with no non-negative edge weights can be considered to represent a discrete distribution over the set of assignments to the Boolean variables. An assignment—or equivalently, the corresponding matched path—is considered to be an

elementary event. The probability of a matched path p is the weight of p divided by the sum of the weights of all matched paths of the WCFLOBDD.

Weight Computation. To sample an assignment from a WCFLOBDD, we first compute the weight corresponding to every exit vertex of every grouping g: the weight of the ith exit vertex of grouping g is the sum of weights of all paths leading to i from the entry vertex of g. This information at every grouping g is computed by recursively calling g's A-connection and B-connection groupings and using the information to compute the information for g.

At level-0 (base case), if g is a `DontCareGrouping`, the weight of the exit vertex is the sum of the weights of the two edges, $(lw + rw)$. If g is a `ForkGrouping`, the weight of exit vertex 1 is lw, and the weight of exit vertex 2 is rw.

Sampling. To sample an assignment a from the probability distribution represented by a WCFLOBDD, half of the assignment, a_A, is sampled recursively from the A-connection, and the other half, a_B, is sampled recursively from one of the B-connections, where $a = a_A \,||\, a_B$. Consider the outermost grouping g and a given exit vertex i. For each middle vertex m of g, the sum of weights of the matched paths from the entry vertex of g to i that passes through m forms a distribution D on g's middle vertices (see Eqn. (M.1), §M.5). To sample a matched path that leads to i we (i) first sample the index (m_{index}) of a middle vertex of g according to D, (ii) recursively sample from g.AConnection with respect to the exit vertex that leads to m_{index}, (iii) recursively sample from g.BConnection[m_{index}] with respect to the exit vertex that leads to i, and (iv) concatenate the sampled paths to obtain the sampled path of g. (Steps (ii) and (iii) can be done in either order.)

At level-0 (base case), if g is a `DontCareGrouping`, the assignment is sampled from "0" and "1" in proportion to the edge weights (lw, rw). If g is a `ForkGrouping`, the assignment "0" or "1" is chosen according to index i.

4.4 Evaluation

Our experiments were designed to answer two questions:

RQ1: Can WCFLOBDDs represent substantially larger functions than WBDDs?

RQ2: In terms of time and space, can WCFLOBDDs outperform WBDDs and CFLOBDDs in a practical domain?

We ran all experiments on an Intel® Xeon® Gold 5218 CPU machine running Ubuntu OS with 31GB RAM, 1000MHz CPU frequency with 64 CPUs. We also set the stack size to "unlimited." The implementation is single-threaded and consists of about 2,500 lines of C++. It also uses Quasimodo (Chapter 5), written in Python.

4.4.1 Experiments to Answer RQ1

For RQ1, we created five synthetic benchmarks, each defining a family of functions whose members (i) have different numbers of variables (all powers of 2), and (ii) could occur as part of a matrix-multiplication sequence in a quantum-circuit simulation:

$$\begin{aligned}
&B1: I_n + X_n \\
&B2: CNOT_n(0, n-1) \times CNOT_n(\tfrac{n}{2}-1, \tfrac{n}{2}) \\
&B3: H_n \times H_n \\
&B4: H_n \times I_n + I_n \times X_n \\
&B5: H_n - H_n
\end{aligned}$$

I_n, X_n, and H_n are the Identity, NOT, and Hadamard matrices, respectively, of size $2^{n/2} \times 2^{n/2}$. $CNOT_n(a, b)$ is a Controlled-NOT matrix of size $2^{n/2} \times 2^{n/2}$. $CNOT(a, b)(\langle x_0, x_1, \ldots, x_{n/2-1}\rangle) = \langle x_0, \ldots, x_a, \ldots, (x_a \oplus x_b), \ldots, x_{n/2-1}\rangle$, for $x \in \{0, 1\}^{n/2}$. We tested our implementation of WCFLOBDDs against WBDDs and CFLOBDDs. We used the MQTDD package Zulehner et al. (2019) (an implementation of WBDDs) and CFLOBDDs implementation Sistla and Reps (2022).

Tab. 4.8 shows the performance of WCFLOBDDs and WBDDs in terms of (i) execution time, and (ii) size. Here "size" means the number of vertices and edges for WCFLOBDDs and CFLOBDDS, and the number of nodes for WBDDs. We find that on the synthetic benchmarks, WCFLOBDDs perform better than WBDDs. Tab. 4.8 also shows the performance of CFLOBDDs on the synthetic benchmarks. We find that WCFLOBDDs are comparable in size to CFLOBDDs on benchmarks B1-B3 and B5, and are smaller than CFLOBDDs on B4. In terms of execution time, WCFLOBDDs are comparable to CFLOBDDs on B1 and B5, and better on B2 and B3. CFLOBDDs perform better than WCFLOBDDs on B4 because B4 involves matrix addition (pointwise addition). In CFLOBDDs, pointwise addition is fast (based on PairProduct), whereas the weight manipulations in WeightedPairProduct (Algs. 67 and 68) have a substantial overhead.

4.4.2 Experiments to Answer RQ2

We applied WBDDs, CFLOBDDs, and WCFLOBDDs to quantum-circuit simulation, which involves exponentially large vectors and matrices. We used MQTDD Zulehner et al. (2019), CFLOBDDs Sistla and Reps (2022), and WCFLOBDDs as back ends to Quasimodo, a quantum-simulation tool that provides a convenient interface for attaching different back-end decision-diagram packages. We used seven standard quantum circuits: Greenberger–Horne–Zeilinger (GHZ), Bernstein–Vazirani (BV), Deutsch–Jozsa (DJ), Simon's algorithm, Quantum Fourier Transform (QFT), Grover's algorithm, and Shor's algorithm.

For each circuit other than GHZ and QFT, a "hidden" string was initially sampled, and the oracle for the circuit was constructed based on the sampled string. For QFT, one of the basis states was sampled and the algorithm changed the basis state into a Fourier-transformed state. For Shor's algorithm, we used the standard $2n + 3$ circuit Beauregard (2002) and report the results for values (N, a), where N is the number to be factored and a is a number used in the phase-estimation procedure. The table in Tab. 4.11 shows the *circuit width* of each benchmark: the total number

of qubits in the circuit as a function of n, the number of qubits for, e.g., the hidden string (reported in column 2 of Tabs. 4.9 and 4.10):

We ran each benchmark 10 times with a 15-minute timeout, starting from a random initial state. Tabs. 4.9 and 4.10 reports average times and sizes of the final state vector. WCFLOBDDs perform better than WBDDs and CFLOBDDs in most cases. The number of qubits that WCFLOBDD can handle is 2,097,152 for GHZ (1× over CFLOBDDs, 64× over WBDDs); 262,144 for BV (1× over CFLOBDDs, 16× over WBDDs); 524,288 for DJ (2× over CFLOBDDs, 32× over WBDDs); 8,192 for Simon's algorithm (1× over CFLOBDDs, 1× over WBDDs) 2,048 for QFT (128× over CFLOBDDs, 1× over WBDDs); and 16 for Grover's algorithm (2× over CFLOBDDs, 1× over WBDDs).

For Simon's algorithm, the number of qubits handled by WCFLOBDDs, WBDDs, and CFLOBDDs is up to 8,192 qubits; however, CFLOBDDs perform better in terms of execution time. For QFT, which is a function whose image is exponentially large, WCFLOBDDs perform better in time than WBDDs (but worse in space), and have substantially better performance in both time and space than CFLOBDDs. For Grover's algorithm and Shor's algorithm, WBDDs perform better than both WCFLOBDDs and CFLOBDDs in both time and space.

In cases when WCFLOBDDs, WBDDs, and CFLOBDDs are all successful, the sizes of the final state vector of WCFLOBDDs and CFLOBDDs are similar, and smaller than WBDDs for GHZ, BV, DJ, and Simon's algorithm. For QFT, Grover's algorithm, and Shor's algorithm, WBDDs represent the final state vector more compactly than WCFLOBDDs and CFLOBDDs.

The evaluation results for CFLOBDD differ from those in §3.9.2.2 because of (i) differences in the machine on which the experiments were run, (ii) differences in the version of the circuits used for some benchmarks. However, in all results reported in Tabs. 4.8, 4.9, and 4.10, WCFLOBDDs, WBDDs, and CFLOBDDs are compared on the same machine.

Benchmark	log(n)	WCFLOBDD		WBDD		CFLOBDD	
		Time (s)	Size	Time (s)	Size	Time (s)	Size
B1	16	**0.001**	442	7.32	65536	**0.001**	**329**
	17	**0.001**	470	14.51	131072	**0.001**	**350**
	18	**0.001**	498	29.23	262144	**0.001**	**371**
	19	**0.001**	526	57.91	524288	**0.001**	**392**
	20	**0.001**	554	117.44	1048576	**0.001**	**413**
	21	**0.001**	582	Timeout		**0.001**	**434**
B2	16	**0.001**	654	7.34	65538	**0.001**	**645**
	17	**0.001**	696	14.56	131074	0.002	**687**
	18	**0.001**	738	29.06	262146	0.003	**729**
	19	**0.001**	780	58.43	524290	0.005	**771**
	20	**0.001**	822	118.44	1048578	0.007	**813**
	21	**0.001**	864	Timeout		0.013	**855**
B3	16	**0.001**	199	7.27	**1**	0.005	197
	17	**0.001**	211	14.4	**1**	0.01	209
	18	**0.001**	223	28.63	**1**	0.018	221
	19	**0.001**	247	57.17	**1**	0.035	233
	20	**0.001**	247	116.52	**1**	0.071	245
	21	**0.001**	259	Timeout		0.142	257
B4	16	0.51	**307**	7.29	32769	**0.003**	647
	17	1.03	**326**	14.52	65537	**0.004**	690
	18	2.13	**345**	28.91	131073	**0.008**	733
	19	4.51	**364**	57.93	262145	**0.017**	776
	20	10	**383**	117.67	524289	**0.04**	819
	21	23.84	**402**	Timeout		0.1	862
B5	16	**0.001**	**83**	7.25	**1**	**0.001**	**83**
	17	**0.001**	**88**	14.38	**1**	**0.001**	**88**
	18	**0.001**	**93**	28.71	**1**	**0.001**	**93**
	19	**0.001**	**98**	57.15	**1**	**0.001**	**98**
	20	**0.001**	**103**	116.77	**1**	**0.001**	**103**
	21	**0.001**	**108**	Timeout		**0.001**	**108**

Figure 4.8: Execution times and sizes of WCFLOBDDs and WBDDs on synthetic benchmarks. (For B3, the WBDD size is 1 because $H \times H = I$, which is handled as a special case in MQTDD.)

Circuit	#Qubits	WCFLOBDD		WBDD		CFLOBDD	
		Time(s)	Size	Time(s)	Size	Time(s)	Size
GHZ	16	0.03	138	**0.01**	**32**	0.02	136
	256	**0.03**	250	0.08	512	**0.03**	**248**
	4096	**0.14**	362	8.45	8192	0.16	**360**
	32768	**1.05**	446	714.97	65536	1.39	**444**
	65536	**2.28**	474	Timeout		3.02	**472**
	2097152	**687.7**	614			760.48	**612**
BV	16	0.03	157	**0.01**	**18**	0.03	156
	256	**0.07**	714	0.12	**258**	0.16	713
	4096	**0.92**	5491	22.83	**4098**	2.8	5490
	16384	**4.52**	16446	447.19	**16386**	13	16445
	65536	**27.08**	58698	Timeout		67.97	**58697**
	262144	**287.87**	218441			515.07	**218440**
DJ	16	0.03	140	**0.01**	**18**	0.03	157
	256	**0.08**	**244**	0.15	258	0.16	269
	4096	**1.01**	**348**	29.35	4098	2.58	381
	16384	**4.59**	**400**	618.48	16386	11.52	437
	65536	**24.71**	**452**	Timeout		56.64	493
	262144	**171.9**	**504**			350.23	549
	524288	**562.21**	**530**			Timeout	
Simon	16	0.07	242	**0.01**	**83**	0.07	327
	256	1.62	**942**	**0.39**	1524	1.68	1524
	4096	124.8	**7042**	93.96	24563	**92.05**	10117
	8192	453.98	**11902**	439.74	49141	**293.73**	17130
QFT	4	0.03	63	**0.01**	**5**	0.02	88
	16	0.03	226	**0.01**	**17**	0.34	69736
	256	**1.39**	3366	1.73	**257**	Timeout	
	2048	**123.28**	26678	777.5	**2049**		

Figure 4.9: Execution times and sizes for quantum-circuit benchmarks for WCFLOBDDs, WBDDs, and CFLOBDDs.

Circuit	#Qubits	WCFLOBDD		WBDD		CFLOBDD	
		Time(s)	Size	Time(s)	Size	Time(s)	Size
Grover	4	0.04	120	**0.01**	**11**	0.04	188
	8	0.16	199	**0.02**	**23**	1.33	1741
	16	8.8	318	**0.51**	**47**	Timeout	
Shor (15, 2)	4	**0.22**	205	0.25	**26**	322.4	2903
Shor (21, 2)	5	2.81	441	**0.19**	**84**	Timeout	
Shor (39, 2)	6	5.29	442	**0.6**	**143**		
Shor (95, 8)	7	Timeout		**684.35**	**1440**		

Figure 4.10: Execution times and sizes for quantum-circuit benchmarks for WCFLOBDDs, WBDDs, and CFLOBDDs. (cont.)

Circuit	*GHZ*	*BV*	*DJ*	Simon	QFT	Grover	Shor
Width	n	$n+1$	$n+1$	$2n$	n	$2n-1$	$2n+3$

Figure 4.11: Circuit widths of the quantum-circuit benchmarks.

Chapter 5: Symbolic Quantum Simulation with Quasimodo

5.1 Introduction

As discussed in Chapter 1, and §2.1, canonical, symbolic representations of Boolean functions—for example, Binary Decision Diagrams (BDDs) Bryant (1986)—have a long history in automated system design and verification. More recently, as seen in Chapters 3 and 4, such data-structures have found exciting new applications in *quantum simulation*. Quantum computers can theoretically solve certain problems much faster than traditional computers, but current quantum computers are error-prone and access to them is limited. The simulation of quantum algorithms on classical machines allows researchers to experiment with quantum algorithms even without access to reliable hardware.

Symbolic function representations are helpful in quantum simulation because a quantum system's state can be viewed as a distribution over an exponential-sized set of basis-vectors (each representing a "classical" state). Such a state, as well as transformations that quantum algorithms typically apply to them, can often be efficiently represented using a symbolic data-structure. Simulating an algorithm then amounts to performing a sequence of symbolic operations.

Currently, there are a small number of open-source software systems that support such *symbolic quantum simulation* Wille et al. (2021); Aleksandrowicz et al. (2021); Cirq Developers (2022); Gray (2018); Tsai et al. (2021). However, as seen in the previous chapters, the underlying symbolic data-structure can have an enormous effect on simulation performance. In this tool paper, we present Quasimodo,[1] an extensible framework for symbolic quantum simulation. Quasimodo is specifically designed for easy extensibility to other backends to make it possible to experiment

[1] Quasimodo is available at `https://github.com/trishullab/Quasimodo.git`.

with a variety of symbolic data-structures. Quasimodo currently supports (i) BDDs Bryant (1986); Fujita et al. (1997); Bahar et al. (1997), (ii) a weighted variant of BDDs Niemann et al. (2016); Viamontes et al. (2004), (Zulehner and Wille, 2020, Ch. 5), (iii) Context-Free-Language Ordered Binary Decision Diagrams CFLOBDDs Chapter. 3, and (iv) Weighted CFLOBDDs Chapter. 4. Quasimodo also has a clean interface that formal-methods researchers can use to plug in new symbolic data-structures, which helps to lower the barrier to entry for formal-methods researchers interested in this area.

Users access Quasimodo through a Python interface. They can define a quantum algorithm as a quantum circuit using 18 different kinds of quantum gates, such as Hadamard, CNOT, and Toffoli gates. They can simulate the algorithm using a symbolic data-structure of their own choosing. Users can sample outcomes from the probability distribution computed through simulation, and can query the simulator for the probability of a specific outcome of a quantum computation over a set of quantum bits (qubits). The system also allows for a form of correctness checking: users are allowed to ask for the set of *all* high-probability outcomes and to check that these satisfy a given assertion.

5.2 Quasimodo's Programming and Analysis Interface

This section provides an overview of Quasimodo from the perspective of a Python API user. A user can define a quantum-circuit computation and check the properties of the quantum state at various points in the computation. This section also explains how Quasimodo can be easily extended to include custom representations of the quantum state.

Example. Fig. 5.1 shows an example of a quantum-circuit computation written using the Quasimodo API. To use the Quasimodo library, one needs to import the package, as shown in line [1]. A user can then create a program that implements a

```python
import quasimodo #python package to import for Quasimodo
epsilon = 1e-8

# number of qubits in the quantum state
numQubits = 2 ** 12
# initialize the quantum state
qs = quasimodo.QuantumState("CFLOBDD", numQubits)
qs.h(0) # Apply Hadamard gate to Qubit 0
for i in range(1, numQubits):
    qs.cx(0, i) # Apply CNOT Gate from Qubit 0 to Qubit i

qubit_mapping = {} # map from qubit number -> desired outcome
for i in range(0, numQubits):
    qubit_mapping[i] = 1

# query probability of outcome as encoded in qubit mapping
prob = qs.prob(qubit_mapping)
if (abs(prob - 0.5)) < epsilon:
    print ("Circuit is correct")
else
    print ("Incorrect circuit")
```

Figure 5.1: An example of a QUASIMODO program that performs a quantum-circuit computation in which the final quantum state is a GHZ state with 4,096 qubits. The program verifies that a measurement of the final quantum state has a 50% chance of returning the all-ones basis-state.

quantum-circuit computation by

- Initializing the quantum state by making a call to `QuantumState` with an argument that selects the desired backend data-structure and the number of qubits in the quantum state. (See line [7].) The example in Fig. 5.1 uses CFLOBDD as the backend simulator, but other data-structures can be used by changing the backend parameter to BDD, WBDD, or WCFLOBDD. `QuantumState` sets the initial quantum state to the all-zeros basis-state.

- Applying single-qubit gates to the quantum state, such as Hadamard (h), Pauli-X (x), T-Gate (t), and others. The qubit to which they are to be applied is

specified by passing the qubit number. (See line [8].)

- Applying multi-qubit gates to the quantum state, such as CNOT (cx), Toffoli (ccx), SWAP (swap), and others. The qubits to which they are to be applied is specified by passing the qubit numbers. (See line [10].)

Note that queries on the quantum state do not have to be made only at the end of the program; they can also be interspersed throughout the circuit-simulation computation.

Quasimodo allows different backend data-structures to be used for representing quantum states. It comes with BDDs Bryant (1986); Fujita et al. (1997); Bahar et al. (1997), a weighted variant of BDDs Niemann et al. (2016); Viamontes et al. (2004), (Zulehner and Wille, 2020, Ch. 5), CFLOBDDs (Chapter 3), and WCFLOBDDs (Chapter 4). Quasimodo also provides an interface for new backend data-structures to be incorporated by users. All four of the standard backends provide compressed representations of quantum states and quantum gates, although—as with all variants of decision diagrams—state representations may blow up as a sequence of gate operations are performed.

Quantum Simulation. Quantum simulation problems can be implemented using Quasimodo by defining a quantum-circuit computation, and then invoking the API function `measure` to sample a basis-vector from the final quantum state. For instance, suppose that the final quantum state is $\begin{bmatrix}0.5 & 0 & 0.5 & 0.5 & 0 & 0 & 0.5 & 0\end{bmatrix}$. Then `measure` would return a string in the set $\{000, 010, 011, 110\}$ with probability 0.25 for each of the four strings.

Verification. As shown in line [17] of Fig. 5.1, Quasimodo provides an API call to inquire about the probability of a specific outcome. The function `prob` takes as its argument a mapping from qubits to $\{0, 1\}$, which defines a basis-vector e of interest,

and returns the probability that the state would be e if a measurement were carried out at that point. It can also be used to query the probability of a set of outcomes, using a mapping of just a subset S of the qubits, in which case `prob` returns the sum of all probabilities of obtaining a state that satisfies S. For example, if the quantum state computed by a 3-qubit circuit over $\langle q_0, q_1, q_2 \rangle$ is $\begin{bmatrix} 0.5 & 0 & 0.5 & 0.5 & 0 & 0 & 0.5 & 0 \end{bmatrix}$, the user can query the probability of states satisfying $q_1 = 1 \wedge q_2 = 0$ by calling $\texttt{prob}(\texttt{1} : \texttt{1}, \texttt{2} : \texttt{0})$, which would returns 0.5 ($= Pr(q_0 = 0 \wedge q_1 = 1 \wedge q_2 = 0) + Pr(q_0 = 1 \wedge q_1 = 1 \wedge q_2 = 0) = (0.5)^2 + (0.5)^2$).

Given a relational specification $R(x, y)$ and a quantum circuit $y = Q(x)$, this feature is useful for verifying properties of the form "$Pr[R(x, Q(x))] > \theta$," where θ is some desired probability threshold for the user's application.

Debugging Quantum Circuits. QUASIMODO additionally provides a feature to query the number of outcomes for a given probability. This feature is especially helpful for debugging large quantum circuits—large in-terms of qubit counts—when most outcomes have similar probabilities.

Consider the case of a quantum circuit whose final quantum state is intended to be $\frac{1}{\sqrt{6}} \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$. One can check if the final quantum state is the one intended by querying the number of outcomes that have probability $\frac{1}{6}$. If the returned value is 6, the user can then check if states 011 and 111 have probability 0 by calling $\texttt{prob}(\{\texttt{0} : \texttt{0}, \texttt{1} : \texttt{1}, \texttt{2} : \texttt{1}\})$ and $\texttt{prob}(\{\texttt{0} : \texttt{1}, \texttt{1} : \texttt{1}, \texttt{2} : \texttt{1}\})$, respectively. The API function for querying the number of outcomes that have probability $\texttt{p} \pm \epsilon$ is $\texttt{measurement_counts}(\texttt{p}, \epsilon)$. One can also query the number of outcomes that have probability $\geq \texttt{p}$ by invoking the function $\texttt{tail_counts}(\texttt{p})$.

QUASIMODO's API provides the methods `get_state()` and `most_frequent()` to obtain the quantum state (as a pointer to the underlying data-structure) and the outcome with the highest probability, respectively.

QUASIMODO not only supports applying the gates of a quantum circuit sequentially, but also first computing various gate-gate operations (either Kronecker product or matrix-multiplication operations) and then apply the resultant gate to the initial quantum state. For example, consider a part of a circuit defined as follows:

```
for i in range(0, n):
    qc.cx(i, n)
```

Instead of applying CNOT (cx) sequentially for every i, one can construct a gate equivalent to $cx_op = \Pi_{i=0}^{n-1} cx(i, n)$ and then apply cx_op to quantum state `qc` as follows:

```
cx_op = qc.create_cx(0, n)
for i in range(1, n):
   tmp = qc.create_cx(i, n)
   cx_op = qc.gate_gate_apply(cx_op, tmp)
qc.apply_gate(cx_op)
```

5.2.1 Extending Quasimodo

The currently supported symbolic data-structures for representing quantum states and quantum gates are written in C++ with bindings for Python. All of the current representations implement an abstract C++ class that exposes (i) `QuantumState`, which returns a state object that represents a quantum state, (ii) eighteen quantum-gate operations, (iii) an operation for gate composition, (iv) an operation for applying a gate—either a primitive gate or the result of gate composition—to a quantum state, and (v) five query operations. Users can easily extend QUASIMODO to add a replacement backend by providing an operation to create a state object, as well as implementations of the seventeen gate operations and three query operations. Currently, the easiest path is to implement the custom representation in C++ as an implementation of the abstract C++ class used by QUASIMODO's standard backends.

5.3 Symbolic Simulation

A symbolic simulation of a quantum circuit-computation Zulehner and Wille (2020); Tsai et al. (2021), Chapters 3, 4 uses a symbolic representation qs of a quantum state and performs operations on qs that correspond to quantum-circuit operations.

- A quantum state of n qubits is a vector of size $2^n \times 1$. Its entries are called *amplitudes*, and the vector represents the probability distribution given by the squares of the absolute values of the amplitudes. In QUASIMODO, CFLOBDDs, WCFLOBDDs, BDDs, and WBDDs are used to represent functions of the form $f : \{0,1\}^n \rightarrow \mathbb{C}$—i.e., f is a vector holding complex amplitudes.

- A quantum gate performs a linear transformation of a quantum state. Quantum-gate application is implemented by using a CFLOBDD, WCFLOBDD, BDD, or WBDD to represent the matrix describing the quantum gate, and performing a matrix-vector multiplication of the gate matrix and the quantum state.

- For CFLOBDDs, WCFLOBDDs, BDDs, and WBDDs, operations like `prob`, `measurement_counts`, and `tail_counts` are implemented as exact operations—i.e., no sampling—via projection and path-counting operations. For CFLOBDDs and BDDs, QUASIMODO computes `prob` via an efficient path-counting operation to obtain the number of paths leading to each terminal value, and then projects the result onto the variables of interest (as specified by the user). QUASIMODO then returns the sum of the probabilities of the remaining paths. In the case of WBDDs and WCFLOBDDs as the backend, QUASIMODO computes the probability of every node ((Zulehner and Wille, 2020, Ch. 5)) or exit vertex of a grouping instead of counting paths. To compute `measurement_counts`, QUASIMODO returns the number of paths that lead to the requested probability value within the provided threshold ϵ. On querying `tail_counts`, QUASIMODO

returns the number of paths that lead to terminal values having probability `prob` $\geq$ `p`, where `p` is the requested probability.

- Once path-counts are computed, a measurement from the CFLOBDD, WCFLOBDD, BDD, or WBDD symbolic representation of a quantum state is a data-structure traversal that can be carried out in time proportional to $\mathcal{O}(\max(\text{number of qubits in the circuit}, \text{size of argument data-structure}))$

Chapter 6: Do CFLOBDDs Actually Make Use of Linear Structure?

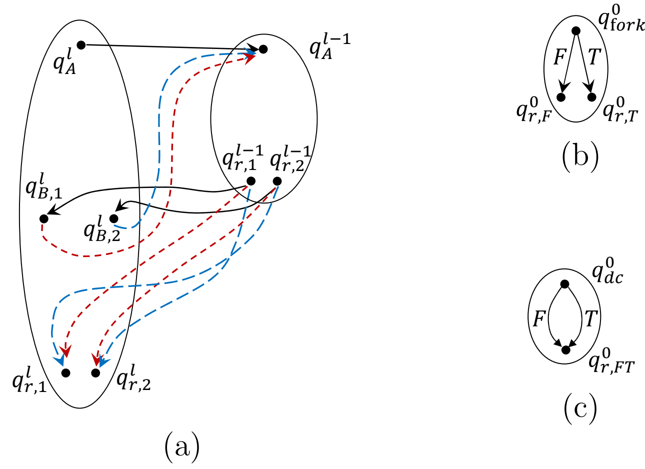
6.1 Introduction

As discussed in Chapter 3, a CFLOBDD is a kind of (non-recursive) hierarchical finite-state machine (HFSM) Alur et al. (2005a), of a special form: every grouping at level l in a CFLOBDD has a "call" to another grouping at level $l-1$ through an AConnection, *followed* by calls to groupings at level $l-1$ through BConnections. The number of BConnections is determined by the number of exits of the AConnection. CFLOBDDs exhibit hierarchical properties through "levels", whereas, in contrast, BDDs do not have a hierarchical structure.

CFLOBDDs can be viewed as going somewhat beyond HFSMs, in that they have some of the features of Nested-Word Automata (NWAs) (or Visibly Pushdown Automata (VPAs)). In particular, like NWAs, CFLOBDDs combine a hierarchical structure with a *linear structure*. §6.1.1 formally defines the linear structure of NWAs and shows how CFLOBDDs are a special case of NWAs.

In Chapter 3, we discussed how the hierarchy in CFLOBDDs provides an exponential compression over BDDs, theoretically (see §3.7) and in practice (see §3.9.2.2). In this chapter, we explore the question of whether CFLOBDDs actually use the linear structure. In particular, §6.4.2 and §6.4.3 discuss how CFLOBDDs can represent functions better than TIDDs because of the lack of linear structure in TIDDs, and present an example of a function f for which there is an exponential gap between CFLOBDDs and TIDDs in terms of representation size—i.e., a CFLOBDD for f can be exponentially smaller than a TIDD for f with the same variable ordering.

In the rest of this subsection, we give an overview of how CFLOBDDs are a restricted form of NWAs, and how they embody hierarchical and linear structure. In §6.2, we introduce Tree-Automata Inspired Decision Diagrams (TIDDs), which have



(a)	call transitions	$(q_A^l, \epsilon, q_A^{l-1}) \in \delta_c$
		$(q_{B,1}^l, \epsilon, q_A^{l-1}) \in \delta_c$
		$(q_{B,2}^l, \epsilon, q_A^{l-1}) \in \delta_c$
	return transitions	$(q_{r,1}^{l-1}, q_A^l, \epsilon, q_{B,1}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_A^l, \epsilon, q_{B,2}^l) \in \delta_r$
		$(q_{r,1}^{l-1}, q_{B,1}^l, \epsilon, q_{r,1}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_{B,1}^l, \epsilon, q_{r,2}^l) \in \delta_r$
		$(q_{r,1}^{l-1}, q_{B,2}^l, \epsilon, q_{r,2}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_{B,2}^l, \epsilon, q_{r,1}^l) \in \delta_r$
(b)	internal transitions	$(q_{\text{fork}}^0, F, q_{r,F}^0) \in \delta_i$
		$(q_{\text{fork}}^0, T, q_{r,T}^0) \in \delta_i$
(c)	internal transitions	$(q_{\text{dc}}^0, F, q_{r,FT}^0) \in \delta_i$
		$(q_{\text{dc}}^0, T, q_{r,FT}^0) \in \delta_i$

Figure 6.1: (a) Encoding of a grouping's A-connection and B-connections as call transitions, and its return edges as return transitions of an NWA. The grouping is the one used to encode the family of Hadamard matrices $\mathcal{H}$. (b) and (c) Encoding of the two kinds of level-0 groupings as internal transitions of an NWA.

hierarchical structure but no linear structure, and explore the relative expressivity of CFLOBDDs and TIDDs.

6.1.1 Nested Words and Nested-Word Automata

Definition 6.1 Alur and Madhusudan (2009)**.** A **nested word** $(w, \rightsquigarrow)$ over alphabet Σ is an ordinary word $w \in \Sigma^*$, together with a **nesting relation** $\rightsquigarrow$ of length $|w|$. $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ such that

- Nesting edges only go forwards: if $i \rightsquigarrow j$ then $i < j$.
- No two edges share a position: for $1 \leq i \leq l$, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$.
- Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.

When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, i is called a **call** position; if $i \rightsquigarrow +\infty$, then i is a **pending call**; otherwise i is a **matched call**, and the unique position j such that $i \rightsquigarrow j$ is called its **return successor**. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, j is a **return** position; if $-\infty \rightsquigarrow j$, then j is a **pending return**, otherwise j is a **matched return**, and the unique position i such that $i \rightsquigarrow j$ is called its **call predecessor**. A position $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.

MatchedNW denotes the set of nested words that have no pending calls or returns. **NWPrefix** denotes the set of nested words that have no pending returns.

A **nested word automaton** (NWA) A is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where

- $\delta_i \subseteq Q \times \Sigma \times Q$ is the transition relation for internal positions.
- $\delta_c \subseteq Q \times \Sigma \times Q$ is the transition relation for call positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation for return positions.

Starting from q_0, an NWA A reads a nested word $nw = (w, \rightsquigarrow)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and $\rightsquigarrow$. If A is in state q when reading input symbol σ at position i in w, and i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. If i is a return position, let k be the call predecessor of i, and q_c be the state A was in just before the transition it made on the kth symbol; A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A **accepts** nw.

The automaton A processes the nested word nw according to a linear order of the characters of nw, along with obeying the hierarchy in nw. As discussed in Alur and Madhusudan (2009), the state propagated along the linear edges (internal transitions) is the same as in the case of a standard word automaton. At a call position, the nesting edge is considered to determine the hierarchical state, and at a return position, the automaton determines the new state based on the linear (internal) and return edges.

Fig. 6.1 illustrates a schema by which a CFLOBDD can be translated to an NWA M. Each matched path through the CFLOBDD corresponds to a nested word in MatchedNW for M. The matched-path principle is obeyed because of the ability of an NWA to "peek" at the state of the most-recent "call" to match a return edge with the appropriate preceding A-connection or B-connection. All transitions taken at a level ≥ 1 are ϵ-transitions (Fig. 6.1a). The only transitions that consume an alphabet symbol are the F and T transitions of the level-0 fork grouping (Fig. 6.1b) and the F and T transitions of the level-0 don't-care grouping (Fig. 6.1c).

Fig. 6.2 shows the nested word that corresponds to the path for the assignment $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$ in the CFLOBDD for the Hadamard matrix H_4, with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$.

For each assignment, the state after a given prefix p is a function of the bindings in p seen so far, making CFLOBDDs depend on the linear order of the assignment. But it is worth pointing out that the hierarchical structure interacts with the linear

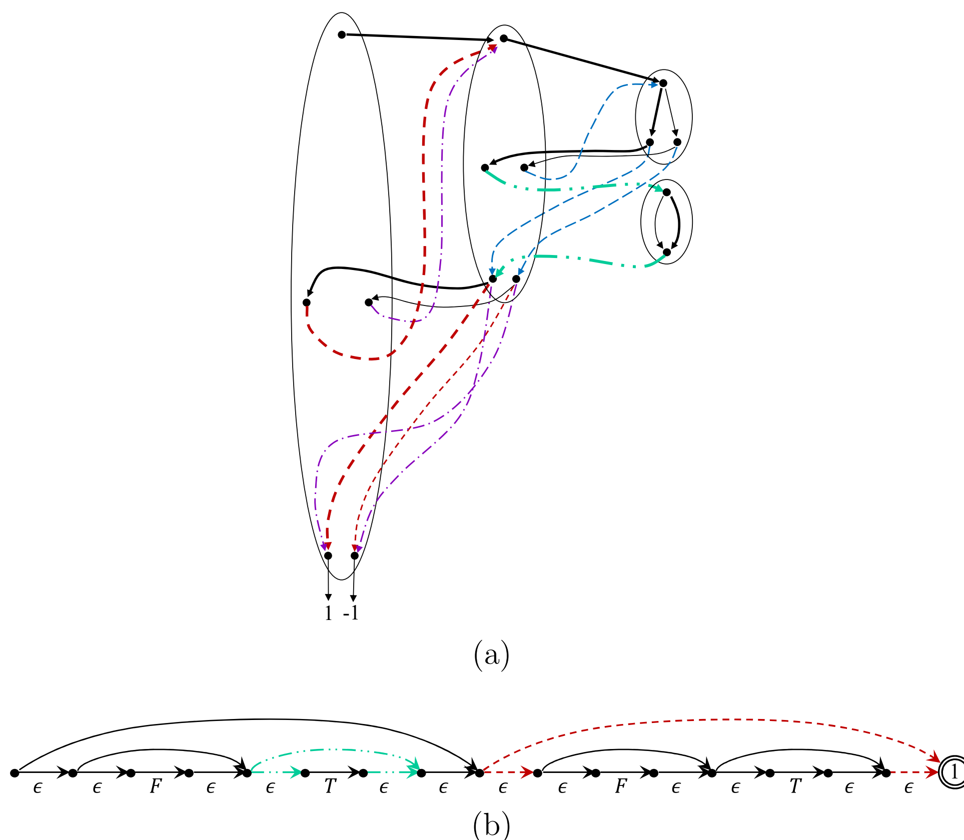


Figure 6.2: (a) The CFLOBDD for the Hadamard matrix H_4, with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$ (repeated from Fig. 3.4a). The matched path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0,3]$ (with value 1), is shown in bold. (b) The nested word for the path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$.

order to allow proto-CFLOBDDs to be shared. For instance, consider the three level-2-to-level-1 calls in Fig. 6.2 (where the three calling contexts are the level-2 grouping's entry vertex, its first middle vertex, and its second middle vertex). It allows the level-1 proto-CFLOBDD to be shared at the three calls—with all three transitioning to the state represented by the entry vertex of the level-1 grouping. At the exit vertices of the level-1 grouping, the various return transitions transfer control to the appropriate states in the level-2 grouping according to the calling context.

To understand whether the functions represented by CFLOBDDs exploit the

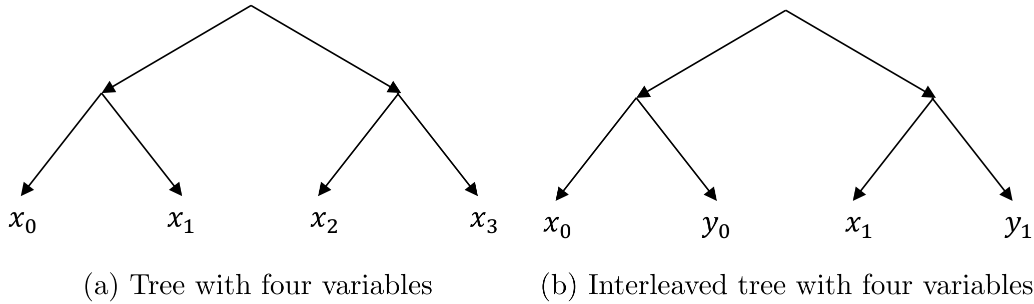
linear structure of words (i.e., assignments), we will create a new decision diagram that is very similar to CFLOBDDs, but without the linear structure. This approach results in the new decision diagrams possessing only the hierarchical properties, and hence can be viewed as a restricted form of deterministic tree automata. More formally, we introduce Tree-automata Inspired Decision Diagrams (TIDD) – acyclic deterministic tree-automata to represent Boolean functions that run on trees whose leaves are values of the different variables of the given function. That is, a tree represents an assignment, and for a pseudo-Boolean function f, the value that the TIDD for f gives to the tree for assignment a is $f(a)$. We want TIDDs to be as close to CFLOBDDs as possible without the linearity in CFLOBDDs; hence, TIDDs also follow an analogue of the "Contextual-Interpretation" principle obeyed by CFLOBDDs: if two sub-functions defined over different groups of variables are equal, then they are represented by the same substructure in the data structure that represents a TIDD. (This point should become clearer in §6.2.)

In particular, we want to understand how deterministic tree automata can represent Boolean functions, and how they compare with CFLOBDDs in terms of the representation size and cost of performing operations.

For the representation of an assignment a, we use a perfect binary tree t_a such that the linear sequence (left-to-right) formed by the leaves of t_a is equal to the assignment a. The tree structure is equivalent to the grammar followed by CFLOBDDs, so the ideas behind TIDDs could be generalized to allow the user to define the tree structure to use (similar to the generalization of CFLOBDDs made in Chapter 7).

However, because CFLOBDDs follow a balanced grammar, TIDDs accept only perfect binary trees, where the tree's leaves contain the function's variables and the internal nodes do not contain any symbols/variables. Fig. 6.3(a) shows the tree for a function over 4 variables – $\langle x_0, x_1, x_2, x_3 \rangle$, and Fig. 6.3(b) shows the tree for a function (e.g., a matrix) over two sets of interleaved variables – $\langle x_0, y_0, x_1, y_1 \rangle$.

We show that CFLOBDDs indeed make use of the linear structure and can



(a) Tree with four variables (b) Interleaved tree with four variables

Figure 6.3: Tree representation of the variables of a function

represent functions more efficiently than TIDDs.

Organization. The rest of the chapter is organized as follows:

- We define TIDDs, a new data structure to represent Boolean functions, relations, matrices, etc. (§6.2).
- We give an overview of the operations using TIDDs (§6.3).
- We discuss examples of two kinds: (i) ones for which TIDDs are as good as CFLOBDDs and exponentially better than BDDs (§6.4.1), (ii) ones that provide intuition of how CFLOBDDs use the linear structure and can represent functions better than TIDDs (§6.4.2), and (iii) example showing exponential gap between CFLOBDDs and TIDDs (with same variable ordering) (§6.4.3)
- We provide some sample evaluations showing CFLOBDDs performing better than TIDDs in practice in the quantum-simulation domain. (§6.5).

6.2 Tree-Automata Inspired Decision Diagrams (TIDDs)

6.2.1 Basic Structure

Because TIDDs are a special version of deterministic finite tree automata that represent Boolean functions, we will define TIDDs in the same way as one defines

a tree automaton, with small differences. Generally, a tree automaton is a tuple $M = (\mathcal{Q}, \mathcal{Q}_f, \mathcal{F}, \Delta)$. M accepts a language of trees $\mathcal{L}$, where $\mathcal{Q}$ is the set of states, $\mathcal{Q}_f$ is the set of final states ($\mathcal{Q}_f \subset \mathcal{Q}$), $\mathcal{F}$ is the set of symbols and Δ represents the transition relation. Because we want to represent functions, there is no notion of "acceptance." All the states are accepted, but every final state has an associated "value." Note that we are only dealing with acyclic deterministic tree automata.

We want to represent functions using TIDDs. An assignment a of the variables of the function is converted to a tree t_a that the tree automaton runs on. If a function f is defined over n Boolean variables, then the corresponding tree t_a is a perfect binary tree of height $\log n$. The leaves of t_a are the Boolean variables, and the internal nodes have one symbol Ω.

Definition 6.2. We define a TIDD as a deterministic, acyclic, bottom-up, tree automaton, divided into "levels" with a tuple $M = (l, \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_l, \mathcal{Q}_f, \mathcal{F}, \Delta, \mathcal{V})$ and the following constraints:

- The number of levels of states in a TIDD is $l + 1$.
- $\mathcal{Q}$ is the set of states. Each Q_i, for $i \in \{0, \ldots, l\}$, is the set of states at level i, called a state-layer.
- $\mathcal{Q}_f$ is the set of final states with $\mathcal{Q}_f = \mathcal{Q}_l$.
- $\mathcal{F}$ is the alphabet, i.e., set of symbols. $\mathcal{F} = \{0, 1, \Omega\}$. **0**, **1** correspond to the symbols at the leaves of the tree, and an implicit symbol – Ω, with arity 2 – corresponds to an internal symbol of the tree.
- Δ is the transition relation. Because the automaton is deterministic, the transition relation Δ becomes a transition function δ of the form: $\delta : \mathcal{Q}_i \times \mathcal{Q}_i \to \mathcal{Q}_{i+1}$, for $i \in \{0, \ldots, l-1\}$. At level-0, $\delta : \mathcal{F} \to Q_0$. We assign a fixed "rejected" state for δ on symbol Ω at level-0—i.e., no leaf is labeled by Ω, and do not explicitly discuss it when defining a TIDD. We refer to an input-output triple

of δ (or input-output pair, in the case of level-0) as either a "transition" or a "hyper-edge" (usually shortened to "edge").

- $\mathcal{V}$ is the value function from every final state ($\mathcal{Q}_f$) to a value $v \in \mathcal{D}$ (some domain), i.e., $\mathcal{V} : \mathcal{Q}_f \rightarrow \mathcal{D}$. Every assignment a of variables of a function f has a corresponding tree t_a with the variables as the leaves of the tree. Hence, the value of the final state reached by the running M on t_a is equal to the value of the function f at a.

TIDDs must also satisfy the following properties.

1. Each assignment has a value (existence of a run): for every assignment-tree t, M does not get stuck: there is a run of M that leads to some final state in $\mathcal{Q}_f$.

2. Uniqueness of the representation of δ:

   (i) At each level i, there is a total order on the states in $\mathcal{Q}_i$. Let $Seq_{\mathcal{Q}_i}$ denote the vector that list the states in that order. The names of states at level i are chosen according to the total order: $[q^i_1, \ldots, q^i_{|Seq_{\mathcal{Q}_i}|}]$.

   (ii) At level 0, the state assigned when the symbol is a 0 is q^0_0; the state assigned when the symbol is a 1 is q^0_1. $Seq_{\mathcal{Q}_0} =_{df} [q^0_0, q^0_1]$.

   (iii) The total order on states at level $i + 1$ equals the order in which they are mentioned in the sequence

   $$Seq_{\mathcal{Q}_{i+1}} = [\delta(q_l, q_r) \mid (q_l, q_r) \in Seq_{\mathcal{Q}_i} \otimes Seq_{\mathcal{Q}_i}]$$

   (where $Seq_{\mathcal{Q}_i} \otimes Seq_{\mathcal{Q}_i}$ is a Kronecker-product operation that pairs sequence elements, and creates a total order on the pairs of states in $\mathcal{Q}_i$).

   (iv) $\mathcal{V}$ is one-to-one and onto $\mathcal{D}$.

   (v) At each level i, any two different states can be distinguished by how they interact with at least one other level-i state via the transition function δ:

   $$\forall q_j, q_k \in \mathcal{Q}_i. \left(q_j \neq q_k \implies \exists q \in \mathcal{Q}_i . \delta(q_j, q) \neq \delta(q_k, q) \vee \delta(q, q_j) \neq \delta(q, q_k)\right).$$

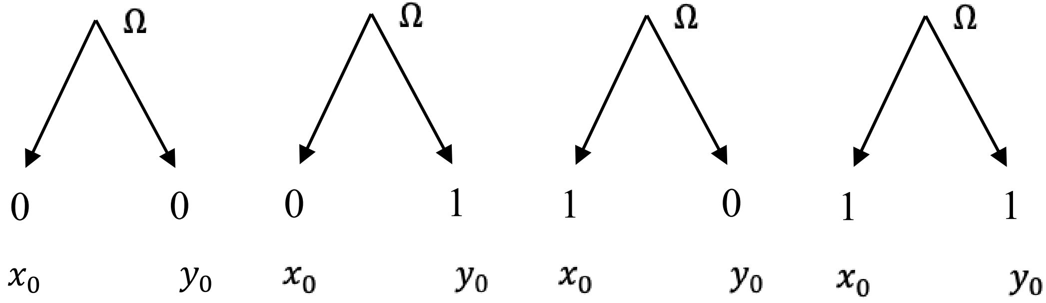
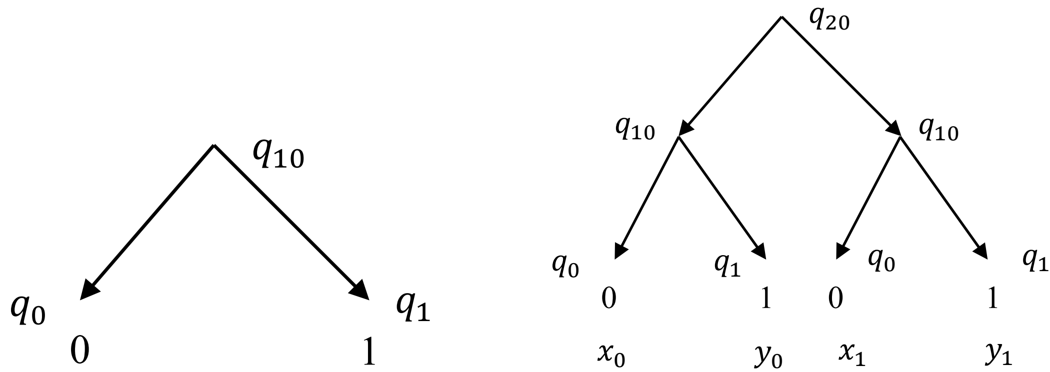


Figure 6.4: The set of trees $\mathcal{T}_{\mathcal{H}}$ representing the assignments of H_2 with two variables x_0, y_0 as the leaves of the tree.



(a) The diagram shows states q_0, q_1, and q_{10}, and transitions $[0 \to q_0,\ 1 \to q_1,\ (q_0, q_1) \to q_{10}]$ of M_{H_2} for the tree representing the assignment $\langle x_0 \mapsto 0, y_0 \mapsto 1\rangle$.

(b) The diagram shows states q_0, q_1, q_{10}, and q_{20}, and transitions $[0 \to q_0,\ 1 \to q_1,\ (q_0, q_1) \to q_{10},\ (q_{10}, q_{10}) \to q_{20}]$ of M_{H_4} for the tree representing the assignment $\langle x_0 \mapsto 0, y_0 \mapsto 1, x_1 \mapsto 0, y_1 \mapsto 1\rangle$.

Figure 6.5: The diagrams show the runs of tree automata (TIDDs) M_{H_2} and M_{H_4} on two trees corresponding to two assignments for the functions (matrices) H_2 and H_4.

Example 6.1. Let us consider the family of Hadamard matrices, discussed in §2.1.1, $\mathcal{H} = \{H_{2^i} \mid i \geq 1\}$, and for $i \geq 1$, $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$, where $H_2 = \begin{array}{c} \\ 0 \\ 1 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 0 & \ \ 1 \end{array} \\ \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{array}$.

H_2 is a function over two Boolean variables – x_0 for row-index variables and y_0 for column-index variables. Hence, H_2 has four assignments corresponding to $\langle x_0, y_0 \rangle = \{00, 01, 10, 11\}$. The set of trees $\mathcal{T}_{\mathcal{H}}$ that represent these assignments are the four binary trees with two leaves shown in Fig. 6.4. The TIDD representation of

H_2 would be a tuple $M_{H_2} = (l = 1, \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1, \mathcal{Q}_f = \mathcal{Q}_1, \mathcal{F} = \{0, 1, \Omega\}, \delta, \mathcal{V})$, with

- $\mathcal{Q}_0 = \{q_0, q_1\}$, $\mathcal{Q}_1 = \{q_{10}, q_{11}\}$
- δ function:

$$\begin{array}{cccc} \multicolumn{2}{c}{\delta(\mathbf{0}) \to q_0} & \multicolumn{2}{c}{\delta(\mathbf{1}) \to q_1} \\ \delta(q_0, q_0) = q_{10} & \delta(q_0, q_1) = q_{10} & \delta(q_1, q_0) = q_{10} & \delta(q_1, q_1) = q_{11} \end{array}$$

- $\mathcal{V}(q_{10}) = 1$, $\mathcal{V}(q_{11}) = -1$

We can observe that M_{H_2} on assignments $\{00, 01, 10\}$ leads to the value $1 = H_2[0, 0] = H_2[0, 1] = H_2[1, 0]$, and assignment $\{11\}$ leads to the value $-1 = H_2[1, 1]$. The TIDD representation of H_2 has 4 states and 6 edges. Fig. 6.5a shows the run of M_{H_2} on the tree with leaves $\langle 0, 1\rangle$, which corresponds to the assignment $[x_0 \mapsto 0, y_0 \mapsto 1]$. We can observe that the states on reading the symbols at the leaves are q_0 for 0 and q_1 for 1, and similarly, the state transitioned to at the next level is $(q_0, q_1) \to q_{10}$.

Let us consider the next matrix in this series $H_4 = H_2 \otimes H_2 = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & H_2 & H_2 \\ 1 & H_2 & -H_2 \end{array} =$

$$\begin{array}{c|cccc} & 00 & 01 & 10 & 11 \\ \hline 00 & 1 & 1 & 1 & 1 \\ 01 & 1 & -1 & 1 & -1 \\ 10 & 1 & 1 & -1 & -1 \\ 11 & 1 & -1 & -1 & 1 \end{array}$$

. The TIDD representation of H_4 is a tuple $M_{H_4} = (l = 2, \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{Q}_f = \mathcal{Q}_2, \mathcal{F} = \{0, 1, \Omega\}, \delta, \mathcal{V})$, with

- $\mathcal{Q}_0 = \{q_0, q_1\}$, $\mathcal{Q}_1 = \{q_{10}, q_{11}\}$, and $\mathcal{Q}_2 = \{q_{20}, q_{21}\}$
- δ function:

$$\begin{array}{cccc} \multicolumn{2}{c}{\delta(\mathbf{0}) \to q_0} & \multicolumn{2}{c}{\delta(\mathbf{1}) \to q_1} \\ \delta(q_0, q_0) = q_{10} & \delta(q_0, q_1) = q_{10} & \delta(q_1, q_0) = q_{10} & \delta(q_1, q_1) = q_{11} \\ \delta(q_{10}, q_{10}) = q_{20} & \delta(q_{10}, q_{11}) = q_{21} & \delta(q_{11}, q_{10}) = q_{21} & \delta(q_{11}, q_{11}) = q_{20} \end{array}$$

- $\mathcal{V}(q_{20}) = 1$, $\mathcal{V}(q_{21}) = -1$

The TIDD representation of H_2 has 6 states and 10 edges. We can observe that $\mathcal{Q}_0, \mathcal{Q}_1$ of M_{H_4} and M_{H_2} are the same. Similarly, the transition function between states of $\mathcal{Q}_0$ and $\mathcal{Q}_1$ of M_{H_4} and M_{H_2} are also the same. Fig. 6.5b shows the run of M_{H_4} on the tree with leaves $\langle 0, 1, 0, 1\rangle$ that corresponds to the assignment $[x_0 \mapsto 0, y_0 \mapsto 1, x_1 \mapsto 0, y_1 \mapsto 1]$. We can observe that the states on reading the symbols at the leaves would be q_0 for 0 and q_1 for 1, similarly, the states transitioned to at the next level would be $(q_0, q_1) \to q_{10}$, and $(q_{10}, q_{11}) \to q_{20}$.

On the same lines, the TIDD representation of H_{2^i} is a tuple $M_{H_{2^i}} = (l = i, \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_i, \mathcal{Q}_f = \mathcal{Q}_i, \mathcal{F} = \{0, 1, \Omega\}, \delta, \mathcal{V})$, with

- $\mathcal{Q}_0 = \{q_0, q_1\}$, $\mathcal{Q}_j = \{q_{j0}, q_{j1}\}$, for $j = 1 \ldots i$.
- δ function, for $j = 1 \ldots i-1$:

$$
\begin{array}{cccc}
\multicolumn{2}{c}{\delta(\mathbf{0}) \to q_0} & \multicolumn{2}{c}{\delta(\mathbf{1}) \to q_1} \\
\delta(q_0, q_0) = q_{10} & \delta(q_0, q_1) = q_{10} & \delta(q_1, q_0) = q_{10} & \delta(q_1, q_1) = q_{11} \\
\delta(q_{10}, q_{10}) = q_{20} & \delta(q_{10}, q_{11}) = q_{21} & \delta(q_{11}, q_{10}) = q_{21} & \delta(q_{11}, q_{11}) = q_{20} \\
\vdots & & & \\
\delta(q_{j0}, q_{j0}) = q_{(j+1)0} & \delta(q_{j0}, q_{j1}) = q_{(j+1)1} & \delta(q_{j1}, q_{j0}) = q_{(j+1)1} & \delta(q_{j1}, q_{j1}) = q_{(j+1)0}
\end{array}
$$

- $\mathcal{V}(q_{i0}) = 1$, $\mathcal{V}(q_{i1}) = -1$

The TIDD representation of H_{2^i} has $2i + 2$ states and $4i + 2$ edges.

Discussion. Note that the trees in Fig. 6.5 do not show the symbol Ω at the internal nodes; we assume it to be implicit. Also, we can observe that, unlike general tree automata, the automata do not have level-0 transitions specific to variables (or, more precisely, to the *position* of a variable in an assignment), but operate solely on the symbols 0 and 1. The reason behind this technique is to make the TIDD representation as similar to CFLOBDDs as possible. If a TIDD were to use a different state for every variable, then the TIDD representation of H_{2^i} would have size $\mathcal{O}(2^i)$, whereas the size of the representation of H_{2^i} used in our work is $\mathcal{O}(i)$.

6.2.2 Canoncity

As shown in Comon et al. (2008), the Myhill–Nerode theorem for tree automata implies the existence and uniqueness (up to isomorphism) of a minimal deterministic tree automaton recognizing a given tree language. We will use the construction following the Myhill-Nerode theorem to show that TIDDs are minimal and thereby canonical.

The Myhill–Nerode equivalence for trees induces a congruence on the set of trees. Two trees are equivalent if and only if they are indistinguishable by all contexts: that is, for any context $C[\,]$, plugging either tree into $C[\,]$ yields trees that are either both accepted or both rejected. Equivalently, a context of a subtree t' in a larger tree t is the surrounding tree with a distinguished hole, and t' may be replaced by another tree in that hole. The minimal deterministic bottom-up tree automaton recognizing a tree language is obtained by taking the Myhill–Nerode equivalence classes as states. Transitions are defined as follows: for a symbol f of arity k and equivalence classes $[t_1], \ldots, [t_k]$, the transition on f maps $([t_1], \ldots, [t_k])$ to the equivalence class of the tree $f(t_1, \ldots, t_k)$.

Formally, two states are equivalent if the trees represented by their equivalence classes are indistinguishable under all contexts. Equivalently, in terms of the transition function, if two states q_1 and q_2 are equivalent — and hence can be merged — then their transition behavior must coincide. That is,

$$\forall q, \quad [\delta(q_1, q)] \equiv [\delta(q_2, q)] \ \wedge \ [\delta(q, q_1)] \equiv [\delta(q, q_2)],$$

where $[\cdot]$ denotes the equivalence class of a state.

We observe that this condition on the transition function has been incorporated into constraint 2(v) that is imposed on the transition structure of TIDDs (page 247). As a result, if the states at level $i + 1$ are minimal, then the states at level i are also minimal, because no two distinct states at level i can satisfy the above equivalence; that is, there are no pairs of states to merge in a TIDD.

At the top-most level l, the number of states is determined by the value function $\mathcal{V}$, which is a bijection onto the domain $\mathcal{D}$. Consequently, the number of states at level l is minimal.

Putting these observations together, we have shown that the states at the top-most level are minimal, and that the transition function of TIDD—being equivalent to that of a minimal deterministic bottom-up tree automaton—ensures that minimality propagates from level $i+1$ to level i. Therefore, TIDDs are minimal by construction.

We now state the canonicity theorem for TIDDs.

Theorem 6.1 (Canonicity of TIDDs)**.** If M_1 and M_2 are level-l TIDDs for the same Boolean function over $n = 2^l$ Boolean variables, and M_1 and M_2 use the same variable ordering, then M_1 and M_2 are isomorphic.

To prove this theorem, we first introduce and discuss two definitions: (i) the *down language* of a state, and (ii) the *down-assignment-language* of a state.

Down Language. Let T represent the set of all possible perfect binary trees with $\mathbf{0}, \mathbf{1}$ as the leaves of the trees. For an automaton A that runs on T, the language accepted by A is

$$L(A) = \{t \in T \mid m_A(t) \in Q_f\}$$

where, $m_A(t)$ is the state obtained by running A on t. In the same way, Guellouma and Cherroun (2016) define the down-language of a state q of the automaton A:

$$L^{\downarrow}(q) = \{t \in T \mid m_A(t) = q\}$$

$L^{\downarrow}(q)$ consists of the set of trees on which automaton A reaches state q.[1]

[1]For a TIDD, $L(A) = T$ because, when run on the tree for an arbitrary assignment, the TIDD reaches *some* final state (Defn. 6.2). In contrast, $L^{\downarrow}(q)$ allows us to make distinctions among different assignment-trees based on the state q reached.

We will now define a slight variant of the down-language. We will define a new term called "down-assignment-language" $L^{\downarrow_f}$. For a Boolean function f over n variables, there exists 2^n perfect binary trees. Every perfect binary tree t can be considered as a one-step-deeper tree over two perfect binary trees t_a and t_b such that t is formed using t_a as the left-child and t_b as the right-child, i.e., $t = t_a||t_b$. Let T represent all such unique binary (sub-)trees of the 2^n perfect binary trees corresponding to the function f. Every tree $t \in T$ has a corresponding word w obtained by sequentially concatenating the leaves of t; let us denote this operation by $w = TW(t)$ and $t = TW^{-1}(w)$. Let T_w represent all such unique words obtained from the trees in T. The "down-assignment-language" $L^{\downarrow_f}$ of a state q of automaton A is defined as

$$L^{\downarrow_f}(q) =_{df} \{w \in T_w \mid m_A(TW^{-1}(w)) = q\}.$$

This definition implies that $L^{\downarrow_f}$ of a state q represents the set of words whose corresponding trees lead to state q on running automaton A. These words are, in fact, partial assignments of the function f. So in other words, if state q belongs to $\mathcal{Q}_i$, where i is the height of the state layer, $L^{\downarrow_f}(q)$ represents the set of Boolean strings of length 2^i that lead to state q on running automaton A.

For a Boolean function f, because TIDDs are deterministic, every $w \in T_w$ and correspondingly, every $t \in T$ belong to the $L^{\downarrow_f}$ of exactly one state, i.e., let $|.|$ represent the length of a word, and let $P(i)$ represent the set of all Boolean-words of length 2^i, then

$$\forall q \in \mathcal{Q}_i, \forall w \in L^{\downarrow_f}(q), |w| = 2^i$$

$$\forall w \in P(i), \exists!\, q \in \mathcal{Q}_i \text{ such that } w \in L^{\downarrow_f}(q)$$

The states (p) of a TIDD at every level-i partition the space of $P(i)$ into p partitions. That is, they satisfy the following properties:

(i) Any two states at the same level have different down-assignment-languages. Let

the TIDD have $l+1$ levels,

$$\forall i \in \{0 \dots l\}, \forall q_j, q_k \in \mathcal{Q}_i, q_j \neq q_k \implies L^{\downarrow_f}(q_j) \cap L^{\downarrow_f}(q_k) = \phi$$

(ii) The union of down-assignment-languages of all states at level-i is equal to $P(i)$.

$$\forall i \in \{0 \dots l\}, \bigcup_{q \in \mathcal{Q}_i} L^{\downarrow_f}(q) = P(i)$$

We will now give the proof for Thm. 6.1.

Proof. Consider two TIDDs M_1 and M_2 that represent the same pseudo-Boolean function.

Base case: Level 0. By condition 2(ii) that is imposed on the transition structure of TIDDs (page 247), M_1 and M_2 have the same level-0 states, namely, q_0^0 when a symbol is 0, and q_1^0 when a symbol is 1, with $Seq_{\mathcal{Q}_0} = [q_0^0, q_1^0]$.

For use in satisfying the hypothesis of the inductive step, note that the corresponding down-assignment-languages are identical:

$$L^{\downarrow_{M_1}}(Seq_{\mathcal{Q}_0}[j]) = L^{\downarrow_{M_2}}(Seq_{\mathcal{Q}_0}[j]), \text{ for } j \in \{0, 1\}.$$

Inductive step: Level i to level $i+1$. Inductive Hypothesis: Assume that at level i the sequences of states in M_1 and M_2 are such that (i) they are the same length; (ii) states in the two TIDDs are named according to position: $Seq_{\mathcal{Q}_i} = [q_0^i, \dots, q^i_{|Seq_{\mathcal{Q}_i}|-1}]$; and (iii) for all $0 \leq j \leq |Seq_{\mathcal{Q}_i}| - 1$, the corresponding down-assignment-languages are identical:

$$L^{\downarrow_{M_1}}(Seq_{\mathcal{Q}_i}[j]) = L^{\downarrow_{M_2}}(Seq_{\mathcal{Q}_i}[j]).$$

Consider the states at level $i+1$. The states of M_1 and M_2 at level $i+1$ are uniquely determined by the transition functions from level i to level $i+1$. Because M_1 and M_2 are minimal, their transition functions coincide. Consequently, the number of states at level $i+1$ in M_1 and M_2 is the same; in particular,

(i) the sequences of states in M_1 and M_2 at level $i+1$ have the same length.

The ordering of states at level $i+1$ is determined by condition 2(iii) on the transition structure of TIDDs (page 247). Specifically, the sequence of states at level $i+1$ is given by

$$Seq_{\mathbb{Q}_{i+1}} = \left[\, \delta(q_l, q_r) \,\middle|\, (q_l, q_r) \in Seq_{\mathbb{Q}_i} \otimes Seq_{\mathbb{Q}_i} \,\right],$$

where $Seq_{\mathbb{Q}_i} \otimes Seq_{\mathbb{Q}_i}$ denotes the Kronecker product of sequences, pairing elements from $Seq_{\mathbb{Q}_i}$. Because $Seq_{\mathbb{Q}_i}$ is identical in M_1 and M_2, it follows that $Seq_{\mathbb{Q}_{i+1}}$ is also identical in both automata, which establishes property (ii).

Consider the down-assignment-language of each state of M_1 and M_2 at level $i+1$. Let $q_j^{i+1} \in Seq_{\mathbb{Q}_{i+1}}$ be a state at level $i+1$, and let

$$\left[(q_{a_1}^i, q_{b_1}^i) \to q_j^{i+1},\, (q_{a_2}^i, q_{b_2}^i) \to q_j^{i+1},\, \ldots,\, (q_{a_k}^i, q_{b_k}^i) \to q_j^{i+1}\right]$$

be the list of transitions leading to q_j^{i+1}, where $q_{a_1}^i, \ldots, q_{a_k}^i, q_{b_1}^i, \ldots, q_{b_k}^i \in Seq_{\mathbb{Q}_i}$. Then the down-assignment language of q_j^{i+1} in M_1 is given by

$$L^{\downarrow M_1}(q_j^{i+1}) = \bigcup_{k'=1}^{k} L^{\downarrow M_1}(q_{a_{k'}}^i) \parallel L^{\downarrow M_1}(q_{b_{k'}}^i),$$

where $\parallel$ denotes the concatenation of words.

Equivalently, for a state $Seq_{\mathbb{Q}_{i+1}}[j]$, the down-assignment language depends only on the down-assignment languages of the states in $Seq_{\mathbb{Q}_i}$, and can be written as

$$L^{\downarrow M_1}(Seq_{\mathbb{Q}_{i+1}}[j]) = \bigcup_{\delta(Seq_{\mathbb{Q}_i}[k_1], Seq_{\mathbb{Q}_i}[k_2]) \to Seq_{\mathbb{Q}_{i+1}}[j]} L^{\downarrow M_1}(Seq_{\mathbb{Q}_i}[k_1]) \parallel L^{\downarrow M_1}(Seq_{\mathbb{Q}_i}[k_2]).$$

An analogous equation holds for $L^{\downarrow M_2}(Seq_{\mathbb{Q}_{i+1}}[j])$. By the induction hypothesis, for all $0 \le j \le |Seq_{\mathbb{Q}_i}|-1$, the corresponding down-assignment languages are identical:

$$L^{\downarrow M_1}(Seq_{\mathbb{Q}_i}[j]) = L^{\downarrow M_2}(Seq_{\mathbb{Q}_i}[j]).$$

Therefore, for all $0 \le j \le |Seq_{\mathbb{Q}_{i+1}}| - 1$, we have

$$L^{\downarrow M_1}(Seq_{\mathbb{Q}_{i+1}}[j]) = L^{\downarrow M_2}(Seq_{\mathbb{Q}_{i+1}}[j]),$$

which establishes property (iii).

Top level l. At the top level l, by the induction hypothesis, the state sequences in M_1 and M_2 have the same length, the same ordering, and identical down-assignment languages. At level l, the value function $\mathcal{V}$ induces a bijection from the states to the domain $\mathcal{D}$. Because $Seq_{\mathcal{Q}_l}$ is identical in M_1 and M_2, the association of each state $Seq_{\mathcal{Q}_l}[j]$ with its value $d \in \mathcal{D}$ is the same in both automata. Therefore, M_1 and M_2 have identical value functions at the top level.

Using the above induction, it follows that the representations of M_1 and M_2 are identical, and hence the TIDD for a pseudo-Boolean function f is a canonical representation of f. □

6.2.3 Relationship of $L^{\downarrow_f}$ to Proto-CFLOBDDs

This section examines the substructures of both TIDDs and CFLOBDDs from the standpoint of their handling of partitions of the set of strings of length 2^i. It shows that (proto-)CFLOBDDs have some additional flexibility that sub-automata of TIDDs lack. This understanding becomes important in §6.4.2 and §6.4.3, where we investigate the differences in the sizes of the TIDD and CFLOBDD that represent a given function f.

As discussed in §3.5 and Appendix §K, a proto-CFLOBDD C at level-i with k exit vertices partitions the space of 2^{2^i} strings of length 2^i, $P(i)$, into k partitions. Consider two proto-CFLOBDDs C and C' at level-i that are a part of a bigger CFLOBDD C_g that represents a function g. Because of the Contextual-Interpretation Principle, in the context of C_g, C and C' can be over the same set of variables, a different set of variables, or both. In any case, C and C' encode different local sub-functions, and thereby partition $P(i)$ differently. Moreover, the partition space of C need not have any correlation with the partition space of C'.

In contrast, because the transitions of a TIDD are not specific to the underlying variables, i.e., the same state is reached when TIDD is run on two trees over different set of variables but with the same assignments, the partition space – the down-

assignment language – of a TIDD is has to be finer than the partition space of every proto-CFLOBDD at the same level. That is, in a TIDD for the function g considered above, there would be a sub-TIDD that has to be prepared to mimic both C and C'. Consequently, the partition space of the sub-TIDD has to be finer than both partition spaces – C and C'.

This property can be stated another way. Let p_c be the number of groupings at level-l; let $[\![P_i]\!]$ denote the partition space of the ith grouping at level-l; let k_i denote the number of partitions in $[\![P_i]\!]$; and let $[\![P_i]\!][k]$ denote the kth partition. If two words w_1, w_2 belong to the same $L^{\downarrow_f}(q)$ for some $q \in \mathcal{Q}_l$ of a TIDD, then they must belong to the same partition in all $[\![P_i]\!]$ partition spaces and if two words w_1, w_2 do not belong to the same $L^{\downarrow_f}(q)$, then there must exist at least one partition space $[\![P_i]\!]$ such that w_1, w_2 do not belong to the same partition in $[\![P_i]\!]$.

$$\forall w_1, w_2 \in L^{\downarrow_f}(q), \text{ for some } q \in \mathcal{Q}, \forall i \in \{1 \ldots p_c\}, \exists k \in \{1 \ldots k_i\}, w_1, w_2 \in [\![P_i]\!][k]$$

and,

$$\forall w_1, w_2, (w_1 \in L^{\downarrow_f}(q_1) \neq w_2 \in L^{\downarrow_f}(q_2), \text{ for some } q_1 \neq q_2 \in \mathcal{Q}_i) \implies$$
$$(\exists i \in \{1 \ldots p_c\}, \exists k_1, k_2 \in \{1 \ldots k_i\}, k_1 \neq k_2 \wedge w_1 \in [\![P_i]\!][k_1] \wedge w_2 \in [\![P_i]\!][k_2])$$

In summary, the set of states at each level-l of the TIDD for a function f corresponds to the coarsest partitioning that is as fine as the partition space of each grouping at level-l of the CFLOBDD that represents f.

6.3 Operations using TIDDs

In this section, we provide an overview of the algorithms for the operations using TIDDs. TIDDs are represented in memory similar to that of CFLOBDDs. Every level i of the TIDD (a state layer) is an object pointing to (1) the state layer at $i-1$, and (2) a list of lists of state numbers (numbered according to a fixed ordering) representing the hyper-edges or the transitions. If the transitions between

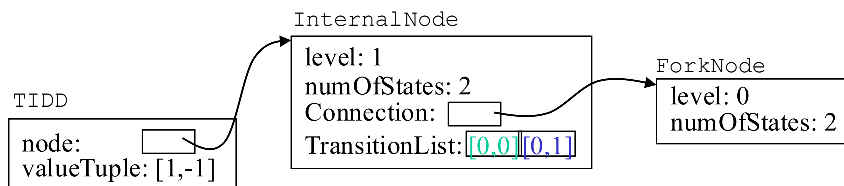


Figure 6.6: Object-oriented representation of TIDD for H_2.

two levels – level-1 and level-2 – are $\delta(q_{10}, q_{10}) \to q_{20}$, $\delta(q_{10}, q_{11}) \to q_{20}$, $\delta(q_{11}, q_{10}) \to q_{21}$, and $\delta(q_{11}, q_{11}) \to q_{21}$, the transitions are represented as: $E = [[0, 0], [1, 1]]$ (a two-dimensional array in row-major order),[2] where $E[0][0] = 0$ represents $\delta(q_{10}, q_{10}) \to q_{20}$, $E[0][1] = 0$ represents $\delta(q_{10}, q_{11}) \to q_{20}$, $E[1][0] = 1$ represents $\delta(q_{11}, q_{10}) \to q_{21}$, and $E[1][1] = 1$ represents $\delta(q_{11}, q_{11}) \to q_{21}$. In general, the transitions between two levels – i and $i+1$ is represented using a 2D array E (a list of lists), where $E[a][b] = c$ represents the transition $\delta(q_{ia}, q_{ib}) \to q_{(i+1)c}$.

The algorithms for TIDDs follow a recursive approach — either bottom-up or top-down — and are similar to the algorithms for tree automata, except that they maintain a total order at every level, thereby ensuring canonicity. Fig. 6.6 shows the object-oriented representation of the TIDD for H_2. As discussed in Ex. 6.1, H_2 has two states at level-0: q_0, q_1, and two states at level-1: q_{10}, q_{11}, with transitions $\delta(q_0, q_0) \to q_{10}$, $\delta(q_0, q_1) \to q_{10}$, $\delta(q_1, q_0) \to q_{10}$, and $\delta(q_1, q_1) \to q_{11}$. The transitions are represented as a TransitionList: $[[0, 0], [0, 1]]$. Every state layer at level-l (> 0) is represented as an `InternalNode(l)`, and the nodes at level-0 are of two kinds: `ForkNode` (with two states), and `DontCareNode` (with one state).

Pragmatics. Techniques such as hash-consing to maintain unique representations, function caching, and equality testing of state layers, and transitions are used and follow a similar pattern to that of CFLOBDDs (and WCFLOBDDs).

[2]Note: in this chapter, unlike the other chapters of the dissertation, we use 0-based array indexing.

Algorithm 34: ConstantTIDD

Input: int l (level), Value v,

Output: TIDD representation of a function with 2^l variables and constant value v

1 **begin**
2 | **return** RepresentativeTIDD(NoDistinctionProtoTIDD(k), $[v]$);
3 **end**

```
Op(Node g) {
  ... // Base case:  Construct appropriate level-0 node
  InternalNode g' = new InternalNode(k); // k = g.level
  // Recursive calls on Op(g.Connection)
  ...
  return RepresentativeNode(g');
}
```

We will define proto-TIDDs that will be useful in a better understanding of the algorithms.

Definition 6.3. A proto-TIDD is similar to a TIDD with all the constraints except for the final value function $\mathcal{V}$. A proto-TIDD is of the form $M' = (l, \mathcal{Q} = \mathcal{Q}_0 \cup \ldots \cup \mathcal{Q}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta)$.

We will now provide a sketch for algorithms for the operations using TIDDs.

6.3.1 Constant Functions

The constant functions $f_0(x) = \lambda x.\bar{0}$ and $f_1(x) = \lambda x.\bar{1}$ have exactly one state at every level. The TIDDs for f_0 and f_1 over n variables have $\log(n)$ levels, with the only difference in $\mathcal{V}$. The TIDD representation of f_0 and f_1, where $n = 2^l$ is a tuple $M = (l = \log(n), \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_l, \mathcal{Q}_f = \mathcal{Q}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta, \mathcal{V})$, with

- $\mathcal{Q}_0 = \{q_0\}$, $\mathcal{Q}_j = \{q_{j0}\}$, for $j = 1 \ldots l$.

Algorithm 35: NoDistinctionProtoTIDD

```
    Input: int l (level)
    Output: Proto-TIDD representation of a function with 2^l variables
 1  begin
 2      if l == 0 then
 3          return RepresentativeDontCareNode;
 4      end
 5      InternalNode g = new InternalNode(l);
 6      g.numOfStates = 1;
 7      g.Connection = NoDistinctionProtoTIDD(l-1);
 8      g.TransitionList = [[0,0]];
 9      return RepresentativeNode(g);
10  end
```

Algorithm 36: FalseTIDD

```
    Input: int l (level)
    Output: TIDD representation of a function with 2^l variables and
            constant value F
 1  begin
 2      return ConstantTIDD(k, F);
 3  end
```

Algorithm 37: TrueTIDD

```
    Input: int l (level)
    Output: TIDD representation of a function with 2^l variables and
            constant value T
 1  begin
 2      return ConstantTIDD(k, T);
 3  end
```

- δ function, for $j = 1 \ldots l - 1$:

$$\delta(\mathbf{0}) \to q_0 \quad \delta(\mathbf{1}) \to q_0$$
$$\delta(q_0, q_0) = q_{10}$$
$$\delta(q_{10}, q_{10}) = q_{20}$$
$$\vdots$$
$$\delta(q_{j0}, q_{j0}) = q_{(j+1)0}$$

- $\mathcal{V}(q_{l0}) = 0$ for f_0, and $\mathcal{V}(q_{l0}) = 1$ for f_1.

Algorithm 38: ProjectionTIDD

```
1 Algorithm ProjectionTIDD(l, i)
    Input: int k (level), int i (index)
    Output: TIDD representing function λx_0, x_1, ..., x_{n-1}.x_i
2   begin
3     if i == 0 then
4       return RepresentativeTIDD(ProjectionProtoTIDD(k,i),
          [T,F]);
5     else
6       return RepresentativeTIDD(ProjectionProtoTIDD(k,i),
          [F,T]);
7     end
8   end
9 end
```

The TIDD-creation operations Algs. 34, 36, and 37 internally call `NoDistinctionProtoTIDD` Alg. 35, which when paired with value tuples [0] and [1] (or $[F]$ and $[T]$) form the TIDDs for the functions f_0 and f_1 respectively.

6.3.2 Projection Functions

A second family of creation operations is *(single-variable) projection functions* of the form $\lambda x_0, x_1, \ldots, x_{n-1}.x_i$, where i ranges from 0 to $n-1$.

Algs. 38 and 39 show the algorithms for creating Projection functions using TIDDs. The level-0 node is always `RepresentativeForkNode` because there need to be two states at level 0. At every level $k > 0$, the transition list depends on $i < 2^{k-1}$. If $i \geq 2^{k-1}$, then transition list: $[[0,0],[1,0]]$, else transition list: $[[0,1],[0,0]]$.

6.3.3 Unary Operations

Scalar Multiplication. Multiplication of a scalar value c with a TIDD M that represents function f can be computed by constructing the TIDD M_c that represents the constant function $\lambda x_0, x_1, \ldots, x_{2^k-1}.c$ (where 2^k is the number of variables of f), and multiplying it with M. The product $M \times M_c$ can be computed using the

Algorithm 39: ProjectionProtoTIDD

```
1  SubRoutine ProjectionProtoTIDD(k, i)
       Input: int k (level), int i (index)
       Output: Node g representing function λx_0, x_1, ..., x_{2^k−1}.x_i
2      begin
3          if k == 0 then
4              return RepresentativeForkNode;
5          else
6              InternalNode g = new InternalNode(k);
7              if i < 2**(k-1) then
8                  g.Connection = ProjectionProtoTIDD(k-1,i);
9                  g.TransitionList = [[0,0],[1,0]];
10                 g.numOfStates = 2;
11             else
12                 i' = i - 2**(k-1);
13                 g.Connection = ProjectionProtoTIDD(k-1,i');
14                 g.TransitionList = [[0,1],[0,0]];
15                 g.numOfStates = 2;
16             end
17             return RepresentativeGrouping(g);
18         end
19     end
20 end
```

construction provided in §6.3.4.

6.3.4 Binary Operations

The binary operation `op` applied to two TIDDs M_1 and M_2, yielding $M = M_1 \, \texttt{op} \, M_2$, is defined via a recursive procedure over the levels of the input TIDDs. The construction consists of two phases: (1) a *cross-product (pair-product)* of the states of M_1 and M_2, followed by (2) a *minimization (reduction)* of the TIDD obtained from the cross-product.

Both the cross-product construction and the subsequent minimization closely follow the standard product and reduction techniques developed in the tree-automata

literature.

Let

$$M_1 = (l, \mathcal{Q} = \mathcal{Q}_0 \cup \cdots \cup \mathcal{Q}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{Q}}, \mathcal{V}_{\mathcal{Q}})$$

and

$$M_2 = (l, \mathcal{P} = \mathcal{P}_0 \cup \cdots \cup \mathcal{P}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}})$$

be two level-l TIDDs. The *cross-product construction* of M_1 and M_2 proceeds in a bottom-up fashion, where states of the resulting TIDD are annotated with pairs of states from the operand TIDDs as metadata.

Let

$$M_{PP} = (l, \mathcal{R} = \mathcal{R}_0 \cup \cdots \cup \mathcal{R}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{R}}, \mathcal{V}_{\mathcal{R}})$$

denote the TIDD obtained from the cross-product construction.

1. **Level 0.** The set of states $\mathcal{R}_0$ is determined by the Cartesian product of $\mathcal{Q}_0$ and $\mathcal{P}_0$, with each state in $\mathcal{R}_0$ corresponding to a pair of states from the two input TIDDs:

$\mathcal{Q}_0$	$\mathcal{P}_0$	$\mathcal{R}_0$	
$\{q_0\}$	$\{p_0\}$	$\{r_0\}$	$= \{(q_0, p_0)\}$
$\{q_0, q_1\}$	$\{p_0\}$	$\{r_0, r_1\}$	$= \{(q_0, p_0), (q_1, p_0)\}$
$\{q_0\}$	$\{p_0, p_1\}$	$\{r_0, r_1\}$	$= \{(q_0, p_0), (q_0, p_1)\}$
$\{q_0, q_1\}$	$\{p_0, p_1\}$	$\{r_0, r_1\}$	$= \{(q_0, p_0), (q_1, p_1)\}$

In an object-oriented representation, the above construction can equivalently be expressed using explicit metadata that records, for each state at a given level, the corresponding pair of operand states:

$\mathcal{Q}_0$	$\mathcal{P}_0$	$\mathcal{R}_0$	Metadata
`DontCareNode`	`DontCareNode`	`DontCareNode`	$[(0, 0)]$
`ForkNode`	`DontCareNode`	`ForkNode`	$[(0, 0), (1, 0)]$
`DontCareNode`	`ForkNode`	`ForkNode`	$[(0, 0), (0, 1)]$
`ForkNode`	`ForkNode`	`ForkNode`	$[(0, 0), (1, 1)]$

The indices appearing in the metadata correspond to $Seq_{\mathcal{R}_0}$ the canonical numbering of level-0 states in $\mathcal{R}$ induced by $Seq_{\mathcal{P}_0}$ and $Seq_{\mathcal{Q}_0}$, which ensures the uniqueness of the representation of $\mathcal{R}$.

2. **Levels $i > 0$.** Each state at level $i > 0$ of the cross-product TIDD corresponds to a pair of states from the operand TIDDs. Formally, the set of states at level i satisfies

$$\mathcal{R}_i \subseteq \mathcal{Q}_i \times \mathcal{P}_i,$$

where a state $r \in \mathcal{R}_i$ is represented by the pair $r = (q, p)$, with $q \in \mathcal{Q}_i$ and $p \in \mathcal{P}_i$. (Equivalently, this pair is stored as metadata associated with r.)

The transition function of the cross-product TIDD is defined component-wise. For states $r_a = (q_a, p_a)$ and $r_b = (q_b, p_b)$, let

$$\delta_{\mathcal{Q}}(q_a, q_b) = q_c \quad \text{and} \quad \delta_{\mathcal{P}}(p_a, p_b) = p_c.$$

Then the transition function $\delta_{\mathcal{R}}$ is given by

$$\begin{aligned} \delta_{\mathcal{R}}(r_a, r_b) &= \delta_{\mathcal{R}}\big((q_a, p_a), (q_b, p_b)\big) \\ &= \big(\delta_{\mathcal{Q}}(q_a, q_b), \delta_{\mathcal{P}}(p_a, p_b)\big) \\ &= (q_c, p_c). \end{aligned}$$

In an object-oriented implementation, this information is maintained as pairs of canonical state identifiers at each level, and is used to construct the states and transitions at the next higher level.

3. **Level l.** At the top-most level, the value function of the cross-product TIDD is defined by combining the values of the corresponding operand states via operation `op`. For a state $r = (q, p) \in \mathcal{R}_l$,

$$\mathcal{V}_{\mathcal{R}}(r) = \mathcal{V}_{\mathcal{R}}(q, p) = \mathcal{V}_{\mathcal{Q}}(q)\, \texttt{op}\, \mathcal{V}_{\mathcal{P}}(p).$$

The next step consists of *reducing* (or *minimizing*) the TIDD resulting from the cross-product construction to obtain a minimal, canonical TIDD. The minimization

procedure follows the standard equivalence-based reduction derived from the Myhill–Nerode theorem, while also generating, at each level i, the canonical naming of states $Seq_{\mathcal{R}_i} = [q_0^i, \ldots, q^i_{|Seq_{\mathcal{R}_i}|-1}]$.

At the top-most level l, states are merged so as to obtain a one-to-one and onto correspondence with the output range of the value function $\mathcal{V}_{\mathcal{R}}$. Let (r, v) denote a state r associated with value v. For example, suppose that the set of state–value pairs at level l of $M_1 \,\texttt{op}\, M_2$ is

$$\{(r_1, v_1), (r_2, v_2), (r_3, v_1)\}.$$

Since states r_1 and r_3 are associated with the same value v_1, they are merged, yielding the minimized set of state–value pairs

$$\{(r_1, v_1), (r_2, v_2)\}.$$

More precisely, the sequence $Seq_{\mathcal{Q}_i}$ induces a total ordering on the states at level i. Rewriting the above example in terms of this ordered representation, if the sequence of state–value pairs at level l is

$$[(r_1, v_1), (r_2, v_2), (r_3, v_1)],$$

then the minimized TIDD contains the sequence

$$[(r_1, v_1), (r_2, v_2)],$$

where state r_3 is merged into r_1, the leftmost occurrence of a state–value pair with value v_1.

In general, when multiple state–value pairs share the same value v, they are merged into a single representative (r, v), chosen to be the leftmost occurrence in the ordered sequence. This merging strategy is analogous to the greedy left-folding used in CFLOBDDs (see §3.3.2 and Appendix §C.)

The equivalence information obtained at level l is then propagated downward through the TIDD. State merging is performed iteratively at each lower level, using the induced equivalence relation on transitions, until level 0 is reached or no further minimization is possible. As discussed in the case of the top-most level l, the merging of a collection of states $\{q_i, q_j, q_k\}$, where the state-sequence at a given level is $[\dots, q_i, \dots, q_j, \dots, q_k, \dots]$, the leftmost instance—here, q_i—becomes the representative of the collection.

6.3.5 Matrix Multiplication

Similar to the discussion in §3.6.4, we use an interleaved variable ordering of the row and column variables to represent matrices.[3] In this section, we present an algorithm for matrix multiplication using TIDDs. The algorithm proceeds in a recursive, bottom-up manner, incrementally constructing and propagating information at each level in the form (q, p, w) (similar to MatMultTuple in §3.6.7), where q and p denote states of the operand TIDDs and w is the associated weight. At the top-most level, each tuple (q, p, w) is resolved to a value, after which the standard reduction procedure is applied to obtain the resulting TIDD.

Let

$$M_1 = (l, \mathcal{Q} = \mathcal{Q}_0 \cup \cdots \cup \mathcal{Q}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{Q}}, \mathcal{V}_{\mathcal{Q}})$$

and

$$M_2 = (l, \mathcal{P} = \mathcal{P}_0 \cup \cdots \cup \mathcal{P}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}})$$

be two TIDDs of the same height l.

Let

$$M_3 = (l, \mathcal{R} = \mathcal{R}_0 \cup \cdots \cup \mathcal{R}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta_{\mathcal{R}}, \mathcal{V}_{\mathcal{R}})$$

denote the TIDD obtained from the bottom-up computation of matrix multiplication on M_1 and M_2.

[3] In the case of vectors, an ascending or descending order of rows (or columns, based on the representation) is used.

1. **Level 1.** Let us assume that $\mathcal{Q}_0$ and $\mathcal{P}_0$ have two states each at level-0. If either operand has only a single state, the construction reduces to a trivial special case of the algorithm described below; we therefore omit a detailed discussion of this case. In this setting, the level-0 states of the product TIDD satisfy $\delta_{\mathcal{R}}(\mathbf{0}) \to r_0$, and $\delta_{\mathcal{R}}(\mathbf{1}) \to r_1$.

   The set of states $\mathcal{R}_1$ is determined as follows:

	$\mathcal{R}_1$
r_{00}	$(\delta_{\mathcal{Q}}(q_0, q_0), \delta_{\mathcal{P}}(p_0, p_0), 1) + (\delta_{\mathcal{Q}}(q_0, q_1), \delta_{\mathcal{P}}(p_1, p_0), 1)$
r_{01}	$(\delta_{\mathcal{Q}}(q_0, q_0), \delta_{\mathcal{P}}(p_0, p_1), 1) + (\delta_{\mathcal{Q}}(q_0, q_1), \delta_{\mathcal{P}}(p_1, p_1), 1)$
r_{10}	$(\delta_{\mathcal{Q}}(q_1, q_0), \delta_{\mathcal{P}}(p_0, p_0), 1) + (\delta_{\mathcal{Q}}(q_1, q_1), \delta_{\mathcal{P}}(p_1, p_0), 1)$
r_{11}	$(\delta_{\mathcal{Q}}(q_1, q_0), \delta_{\mathcal{P}}(p_0, p_1), 1) + (\delta_{\mathcal{Q}}(q_1, q_1), \delta_{\mathcal{P}}(p_1, p_1), 1)$

   Each state in $\mathcal{R}_1$ is represented as a formal sum of weighted triples (q, p, w), where $q \in \mathcal{Q}_1$, $p \in \mathcal{P}_1$, and $w \in \mathbb{N}$ is the associated weight.

   Semantics (Matrix Multiplication). Each triple (q, p, w) corresponds to a contribution of weight w to the product of the matrix entries represented by states q and p. Specifically, for fixed row index i and column index j, the state r_{ij} encodes the summation

   $$r_{ij} \equiv \sum_{k \in \{0,1\}} \big(\delta_{\mathcal{Q}}(q_i, q_k), \delta_{\mathcal{P}}(p_k, p_j), 1\big),$$

   which mirrors the standard matrix multiplication rule

   $$(C)_{ij} = \sum_k (A)_{ik} \cdot (B)_{kj}.$$

   Addition of triples. Given two triples (q_a, p_a, w_a) and (q_b, p_b, w_b), their sum is defined as

   $$(q_a, p_a, w_a) + (q_b, p_b, w_b) = \begin{cases} (q_a, p_a, w_a + w_b), & \text{if } q_a = q_b \text{ and } p_a = p_b, \\ \text{two distinct triples}, & \text{otherwise.} \end{cases}$$

When merging triples with identical state pairs, the representative (q, p) is chosen according to the fixed total order used for canonicalization.

Thus, at each level of the construction, the algorithm maintains a unique set of state pairs (q, p) together with an accumulated weight, exactly capturing the additive structure of matrix multiplication within the TIDD framework.

Note that states r_{00}, r_{01}, r_{10}, and r_{10} may be merged whenever their associated sets of triples are identical.

2. **Levels $i > 0$.** Each state at level $i > 0$ is represented as a formal sum of weighted triples of the form

$$r = (q_{j_1}, p_{j_1}, w_{j_1}) + \cdots + (q_{j_k}, p_{j_k}, w_{j_k}),$$

where $q_{j_\ell} \in \mathcal{Q}_i$, $p_{j_\ell} \in \mathcal{P}_i$, and w_{j_ℓ} denotes the associated weight.

Let

$$r_a = (q_{a_1}, p_{a_1}, w_{a_1}) + \cdots + (q_{a_{k_1}}, p_{a_{k_1}}, w_{a_{k_1}}),$$
$$r_b = (q_{b_1}, p_{b_1}, w_{b_1}) + \cdots + (q_{b_{k_2}}, p_{b_{k_2}}, w_{b_{k_2}})$$

be two such states. The transition function $\delta_{\mathcal{R}}$ is defined by distributing over the sums and combining all pairwise contributions:

$$\delta_{\mathcal{R}}(r_a, r_b) = \sum_{u=1}^{k_1} \sum_{v=1}^{k_2} \big(\delta_{\mathcal{Q}}(q_{a_u}, q_{b_v}), \delta_{\mathcal{P}}(p_{a_u}, p_{b_v}), w_{a_u} \cdot w_{b_v}\big).$$

As in the level-1 construction, the resulting collection of triples is canonicalized by merging triples with identical state pairs (q, p) and accumulating their weights. Consequently, at each level, only states with distinct sets of weighted triples are maintained.

3. **Level l.** At the top-most level-l, each state of the form

$$r = (q_{j_1}, p_{j_1}, w_{j_1}) + \cdots + (q_{j_k}, p_{j_k}, w_{j_k}),$$

is resolved and mapped to a fixed value as:

$$\mathcal{V}_{\mathcal{R}}(r) = \mathcal{V}_{\mathcal{R}}((q_{j_1}, p_{j_1}, w_{j_1}) + \cdots + (q_{j_k}, p_{j_k}, w_{j_k})) = \sum_{u=0}^{k} \mathcal{V}_{\mathcal{Q}}(q_u) \times \mathcal{V}_{\mathcal{P}}(p_u) \times w_u$$

The unique states and their associated values are computed as shown at the top level. States with the same value are merged—with the left-most instance chosen as the representative—and this information is propagated downward using the TIDD reduction algorithm discussed earlier.

6.3.6 Sampling

A TIDD with only non-negative values in the range of $\mathcal{V}$ can be considered to represent a discrete distribution over the set of assignments to the Boolean variables. An assignment—or equivalently, the corresponding binary tree—is considered to be an elementary event. The probability of the assignment (or tree) p, on which TIDD runs, is the value of p divided by the sum of the values of all assignments (trees) of TIDD.

We begin by computing $|L^{\downarrow_f}(q)|$, the size of $L^{\downarrow_f}(q)$ for every state q,where $L^{\downarrow_f}(q)$ denotes the set of Boolean assignments that lead to state q. We also call $|L^{\downarrow_f}(q)|$ the *path count* for q.

Path counting. For each state q, the path count $|L^{\downarrow_f}(q)|$ can be computed recursively in a bottom-up fashion. Let

$$\{\delta(q_{a_1}, q_{b_1}) \rightarrow q,\ \delta(q_{a_2}, q_{b_2}) \rightarrow q,\ \ldots,\ \delta(q_{a_k}, q_{b_k}) \rightarrow q\}$$

be the set of transitions whose target is q. Then,

$$|L^{\downarrow_f}(q)| = \sum_{u=1}^{k} |L^{\downarrow_f}(q_{a_u})| \cdot |L^{\downarrow_f}(q_{b_u})|.$$

At level 0, if the TIDD contains two states, then the path count of each state is 1; otherwise, the unique state has path count 2.

Sampling. Let $\mathcal{T}$ be a TIDD that represents a (possibly weighted) distribution of the outputs of the Boolean function. We define a top-down procedure to sample an assignment $a \in \{0,1\}^n$ from the probability distribution induced by $\mathcal{T}$.

Top-level sampling. At the top-most level l, let $\mathcal{R}_l$ be the set of states. Each state $q \in \mathcal{R}_l$ is assigned a weight

$$W(q) \;\coloneqq\; \mathcal{V}(q) \cdot |L^{\downarrow_f}(q)|,$$

where $\mathcal{V}(q)$ is the value associated with q and $|L^{\downarrow_f}(q)|$ denotes the number of assignments leading to q. These weights induce a probability distribution

$$\Pr[q] \;=\; \frac{W(q)}{\sum_{q' \in \mathcal{R}_l} W(q')}.$$

A state q is sampled according to this distribution.

Recursive step. Given a sampled state q at level $i > 0$, let

$$\mathcal{T}(q) = \{|L^{\downarrow_f}(q)|\}$$

denote the set of transitions whose target is q. One such transition $\delta(q_j, q_k) \to q$ is selected based on the probability.

$$\Pr[\delta(q_j, q_k) \to q] = \frac{|L^{\downarrow_f}(q_j)| \cdot |L^{\downarrow_f}(q_k)|}{|L^{\downarrow_f}(q)|}.$$

The sampling procedure is then invoked recursively on states q_j and q_k, yielding assignments a_j and a_k, respectively. The assignment returned for state q is defined as the concatenation

$$a \;=\; a_j \,\|\, a_k.$$

Base case (level 0). At level 0, each state corresponds to an assignment of a single Boolean variable. The sampling behavior depends on the number of paths to a state at level 0. If a level-0 state q has two transitions corresponding to assignments 0 and

1, then one of the two assignments is chosen uniformly at random. If state q has only one transition, then the assignment corresponding to the transition – 0 or 1 – is chosen.

In both cases, the base case returns a length-1 assignment, which is then used in the recursive construction of higher-level assignments.

This recursive procedure samples assignments according to the probability distribution represented by the TIDD.

6.4 Understanding TIDDs, CFLOBDDs, and BDDs through examples

In this section, we will discuss (1) functions that are efficiently represented by TIDDs—i.e., have similar efficiency to CFLOBDDs—and can be exponentially smaller than any BDD for the function (§6.4.1), (2) give intuition why CFLOBDDs represent functions better than TIDDs because of the lack of linear structure in TIDDs (§6.4.2), and (3) discuss an example where there is an exponential gap between CFLOBDDs and TIDDs in terms of representation size (§6.4.3)—the CFLOBDD for a function f can be exponentially smaller than any TIDD for f.

6.4.1 Relations Efficiently Represented by TIDDs

In this section, we prove that there exists an exponential separation between TIDDs and BDDs, using functions where the TIDD is of a similar size to the functions' CFLOBDDs. We establish this result using two relations that can be efficiently represented by TIDDs and CFLOBDDs, but not by BDDs. These examples are taken from §3.7.1 and §3.7.2, where CFLOBDDs show exponential compression over BDDs.

6.4.1.1 The Hadamard Relation H_n

We will use the Hadamard Relation H_n (discussed in §3.7.2) to show that TIDDs efficiently represent (some) functions better than BDDs.

Theorem 6.2 Exponential separation for the Hadamard relation**.** The Hadamard Relation $H_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \rightarrow \{1,-1\}$ between variable sets $(x_0 \cdots x_{n/2})$ and $(y_0 \cdots y_{n/2})$, where $n = 2^l$, can be represented by a TIDD of size $\mathcal{O}(\log n)$, similar to that of the CFLOBDD for H_n. In contrast, any BDD that represents H_n requires $\Omega(n)$ nodes.

Proof. TIDD Claim. As shown in Ex. 6.1, each matrix $H_n \in \mathcal{H}$, where $n = 2^l$ can be represented by a TIDD with $\mathcal{O}(l)$ vertices and edges—i.e., of size $\mathcal{O}(\log n)$.

BDD Claim. Regardless of the variable ordering, the BDD representation for H_n requires at least n nodes, one node for each variable in the argument, as discussed in Thm. 3.4. □

6.4.1.2 The Equality Relation EQ_n

We will use the Equality Relation EQ_n (discussed in §3.7.1) to show that TIDDs efficiently represent (some) functions better than BDDs.

Definition 6.4. The equality relation $EQ_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \rightarrow \{0,1\}$ on variables $(x_0 \cdots x_{n/2-1})$ and $(y_0 \cdots y_{n/2-1})$ is the relation $EQ_n(X,Y) \stackrel{\text{def}}{=} \Pi_{i=0}^{n/2-1}(x_i \Leftrightarrow y_i) = \Pi_{i=0}^{n/2-1}(\bar{x}_i\bar{y}_i \vee x_i y_i)$.

Theorem 6.3 Exponential separation for the equality relation**.** For $n = 2^l$, where $l \geq 1$, EQ_n can be represented by a TIDD of size $\mathcal{O}(\log n)$. In contrast, a BDD that represents EQ_n requires $\Omega(n)$ nodes.

Proof. TIDD Claim. We claim that with the interleaved-variable ordering $\langle x_0, y_0, \ldots, x_{n/2-1}, y_{n/2-1} \rangle$, the TIDD representation of EQ_n uses $\mathcal{O}(\log n)$ states and edges.

With the interleaved variable ordering, the TIDD representation of EQ_{2^l}, where $n = 2^l$ is a tuple $M_{EQ_{2^l}} = (l = l, \mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_l, \mathcal{Q}_f = \mathcal{Q}_l, \mathcal{F} = \{0, 1, \Omega\}, \delta, \mathcal{V})$, with

- $\mathcal{Q}_0 = \{q_0, q_1\}$, $\mathcal{Q}_j = \{q_{j0}, q_{j1}\}$, for $j = 1 \ldots l$.
- δ function, for $j = 1 \ldots l - 1$:

$$\begin{array}{cccc}
\multicolumn{2}{c}{\delta(\mathbf{0}) \rightarrow q_0} & \multicolumn{2}{c}{\delta(\mathbf{1}) \rightarrow q_1} \\
\delta(q_0, q_0) = q_{10} & \delta(q_0, q_1) = q_{11} & \delta(q_1, q_0) = q_{11} & \delta(q_1, q_1) = q_{10} \\
\delta(q_{10}, q_{10}) = q_{20} & \delta(q_{10}, q_{11}) = q_{21} & \delta(q_{11}, q_{10}) = q_{21} & \delta(q_{11}, q_{11}) = q_{21} \\
\vdots & & & \\
\delta(q_{j0}, q_{j0}) = q_{(j+1)0} & \delta(q_{j0}, q_{j1}) = q_{(j+1)1} & \delta(q_{j1}, q_{j0}) = q_{(j+1)1} & \delta(q_{j1}, q_{j1}) = q_{(j+1)1}
\end{array}$$

- $\mathcal{V}(q_{l0}) = 1$, $\mathcal{V}(q_{l1}) = 0$

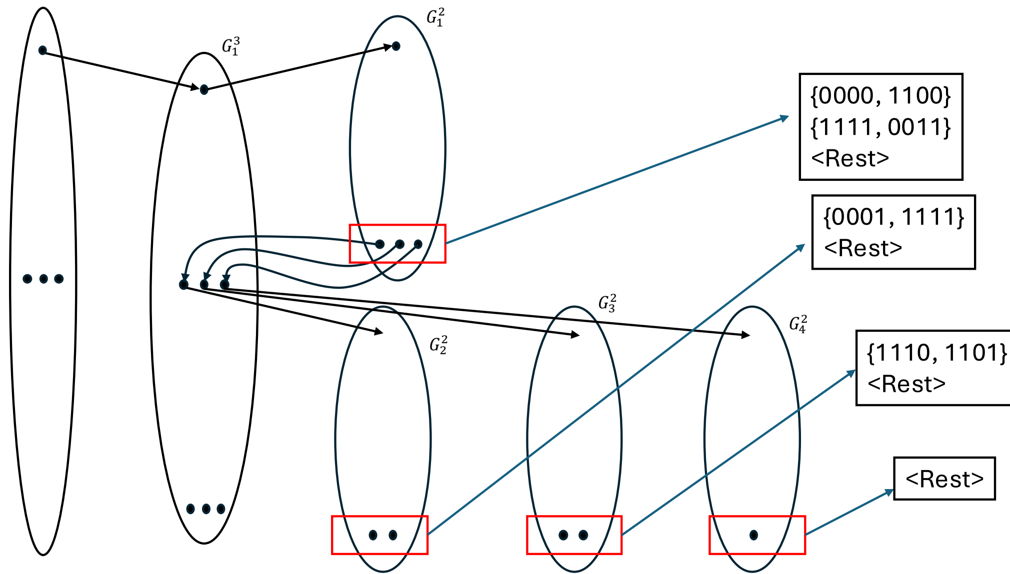
We observe that $M_{EQ_{2^i}}$ represents EQ_{2^i} and has $2i + 2$ states and $4i + 2$ edges. Therefore, the TIDD representation of EQ_n is of size $\mathcal{O}(\log n)$.

BDD Claim. Regardless of the variable ordering, the BDD representation for EQ_n requires at least n nodes, one node for each variable in the argument, as discussed in Thm. 3.3. □

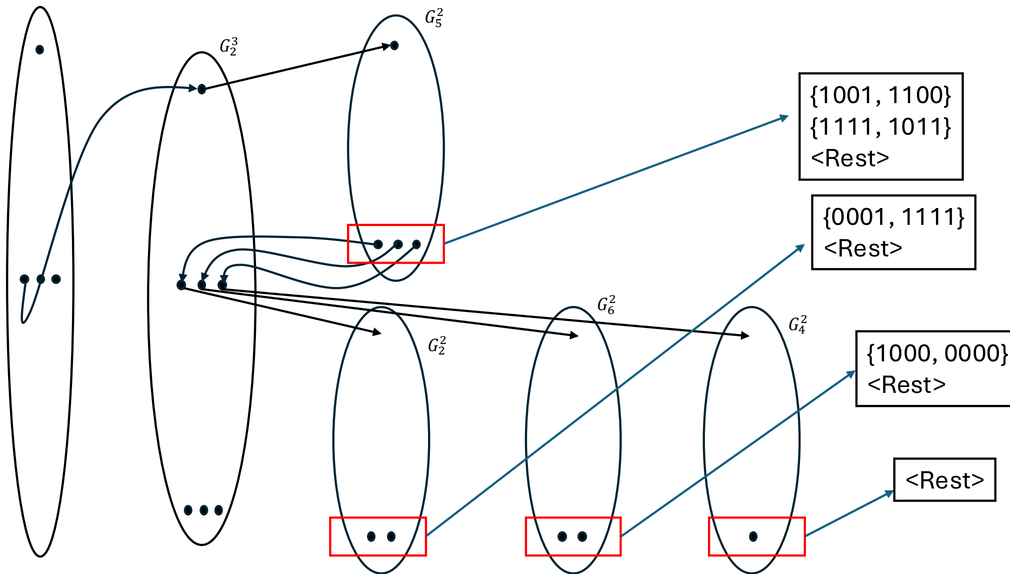
6.4.2 Examples Showing the Use of Linearity in CFLOBDDs

To understand how linearity in CFLOBDDs helps it to represent functions more efficiently than TIDDs, let us take a look at the example shown in Fig. 6.7.

Fig. 6.7 shows a proto-CFLOBDD that has another proto-CFLOBDD G_1^3 at level-3 as AConnection and G_2^3 at level-3 as the first BConnection. Edges not necessary for the discussion are omitted to remove clutter. Let us understand G_1^3 in detail. G_1^3 has an AConnection G_1^2 and three BConnections: G_2^2, G_3^2, and G_4^2. Each G_x^2 is a proto-CFLOBDD at level-2 over 4 variables that partition $P(4)$ into distinct



(a) Diagram showing a proto-CFLOBDD with AConnection to a proto-CFLOBDD whose AConnection and BConnections at level-2 partition $P(4)$ as shown.



(b) Diagram showing a proto-CFLOBDD with a BConnection to a proto-CFLOBDD whose AConnection and BConnections at level-2 partition $P(4)$ as shown.

Figure 6.7: Diagrams showing a proto-CFLOBDD with an AConnection (Fig. 6.7a) and a BConnection (Fig. 6.7b) to proto-CFLOBDDs that partition the space of strings differently; edges not necessary for the discussion are omitted to remove clutter.

partitions:

$$\begin{aligned}
[\![P_{G_1^2}]\!] &= [\{0000, 1100\}, \{1111, 0011\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_2^2}]\!] &= [\{0001, 1111\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_3^2}]\!] &= [\{1110, 1101\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_4^2}]\!] &= [\langle \texttt{Rest} \rangle] = P(4)
\end{aligned}$$

where $\langle \texttt{Rest} \rangle$ means $P(4) \setminus$ all explicitly listed sets. In the case of $[\![P_{G_4^2}]\!]$, $\langle \texttt{Rest} \rangle = P(4)$. We observe that there are four unique groupings at level-2 and 12 edges between groupings at level-2 and level-3, including 5 return edges omitted from the diagram of G_3^1.

As discussed in §6.2.3, the states of TIDD at level-2 would depend on the partitions of all the groupings of the CFLOBDD at level-2. For now, let us consider the partitions seen above and deduce the states and $L^{\downarrow_f}$ of the states just based on the partitions seen so far. Based on the discussion in §6.2.3, the states of the TIDD at level-2 should satisfy the property: If two words belong to the same partition in all the partition spaces of CFLOBDD groupings, they belong to the same state's down-assignment-language in TIDD. Similarly, if there exists at least one partition space where two words do not belong to the same partition, then they must belong to the down-assignment-language of different states. Using this insight, let us list the states at level-2 of the TIDD representing this function and their corresponding $L^{\downarrow_f}$.

$$\begin{aligned}
q_1 &: L^{\downarrow_f}(q_1) = \{0000, 1100\} \\
q_2 &: L^{\downarrow_f}(q_2) = \{1111\} \\
q_3 &: L^{\downarrow_f}(q_3) = \{0011\} \\
q_4 &: L^{\downarrow_f}(q_4) = \{0001\} \\
q_5 &: L^{\downarrow_f}(q_5) = \{1110, 1101\} \\
q_6 &: L^{\downarrow_f}(q_6) = \langle \texttt{Rest} \rangle
\end{aligned}$$

The above partition satisfies the property discussed. This leads to a TIDD having 6 states and 25 edges at level 2, which is more than the size in the case of a CFLOBDD representation. This is because, in CFLOBDDs, the BConnections are constructed based on the exit vertices of the AConnection at a given level, thereby every grouping encodes a local (sub-)function and partitions the space of $P(2^{2^i})$ indi-

vidually. As a result, the number of groupings and, particularly, the number of edges to represent this information is vastly reduced.

This observation, that the groupings at a given level l can have different partitions, provides intuition for why the linear structure inherent in CFLOBDDs (and not inherited by TIDDs) enables a large function to be decomposed into smaller subfunctions, each of which can be represented efficiently. We now extend this example to illustrate this intuition further.

Let us now consider the proto-CFLOBDD shown in Fig. 6.7b. The same proto-CFLOBDD at level-4 has a BConnection to a proto-CFLOBDD G_2^3 at level-3. G_2^3 has an AConnection to G_5^2 and three BConnections: G_2^2 (reuse from Fig. 6.7a), G_6^2, and G_4^2 (reuse from Fig. 6.7a). Each of these groupings forms the following partition spaces:

$$\begin{aligned}
[\![P_{G_5^2}]\!] &= [\{1001, 1100\}, \{1111, 1011\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_2^2}]\!] &= [\{0001, 1111\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_6^2}]\!] &= [\{1000, 0000\}, \langle \texttt{Rest} \rangle] \\
[\![P_{G_4^2}]\!] &= [\langle \texttt{Rest} \rangle] = P(4)
\end{aligned}$$

There are four unique groupings at level-2 and 12 edges between groupings at level-2 and level-3, considering only the ones in Fig. 6.7b. The total number of unique groupings and edges at level-2 among AConnection and first BConnection of the top-level CFLOBDD (i.e., considering Figs. 6.7a and 6.7b) is 6 groupings at level-2 and 24 edges.

The new states of TIDD at level-2 would be:

$$
\begin{array}{rcl}
q'_1 & : & L^{\downarrow_f}(q'_1) = \{0000\} \\
q'_2 & : & L^{\downarrow_f}(q'_2) = \{1111\} \\
q'_3 & : & L^{\downarrow_f}(q'_3) = \{0011\} \\
q'_4 & : & L^{\downarrow_f}(q'_4) = \{0001\} \\
q'_5 & : & L^{\downarrow_f}(q'_5) = \{1110, 1101\} \\
q'_6 & : & L^{\downarrow_f}(q'_6) = \langle \texttt{Rest} \rangle \\
q'_7 & : & L^{\downarrow_f}(q'_7) = \{1100\} \\
q'_8 & : & L^{\downarrow_f}(q'_8) = \{1001\} \\
q'_9 & : & L^{\downarrow_f}(q'_9) = \{1011\} \\
q'_{10} & : & L^{\downarrow_f}(q'_1) = \{1000\}
\end{array}
$$

We can observe that with the addition of new partition spaces, the states of TIDD increase from 6 to 10. Particularly, the state q_1 bifurcates into q'_1, q'_7 and q_6 bifurcates into $q'_6, q'_8, q'_9, q'_{10}$. Thereby, the number of states at level-2 of the TIDD is 10, and the number of edges is 81. This is significantly higher than the number of groupings and edges in the CFLOBDD. We also observe that the rate of growth of the states of a TIDD, and especially the edges, grows quite fast.

This example illustrates that the linear structure of CFLOBDDs enables a large function to be decomposed into smaller subfunctions that each admit efficient representations, thereby yielding an efficient representation overall—a decomposition that is not possible in a TIDD.

6.4.3 Exponential Separation between CFLOBDDs and TIDDs

In this section, we present an example showing that CFLOBDDs can be exponentially more succinct than TIDDs under the same variable ordering. We emphasize that this separation is relatively weak, because it does not rule out the existence of a more favorable variable ordering for TIDDs. However, the result serves to highlight how the linear structure of CFLOBDDs enables exponentially more succinct representations than TIDDs.

The function that we use can be thought of as taking a Boolean matrix as input and returning 1 if and only if all elements on the anti-diagonal are 0. That is,

the function returns 1 for a matrix of the form

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 & \cdot \\ \cdot & \cdot & 0 & \cdot & \cdot \\ \cdot & 0 & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

As defined in Defn. 6.5, the matrix entries are presented in row-major order.

Definition 6.5. Function $h_n : \{0,1\}^{n^2} \rightarrow \{0,1\}$ on variables $X = (X_0, \ldots, X_{n-1}) = (x_0 \ldots x_{n^2-1})$ – i.e., where $X_i = (x_{i*n} \ldots x_{i*n+n-1})$ is defined as $h_n(X) \stackrel{\text{def}}{=} \bigwedge_{i=0}^{n-1} f_i(X_i)$, with

$$f_i(X_i) = \begin{cases} 1 & i^{th} \text{ bit of } X_i\text{: } x_{i*n+n-1-i} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Function f_i tests the ith row for the pattern "Is the ith entry of the row equal to 0?"

Theorem 6.4 Exponential separation**.** For $n = 2^l$, where $l \geq 1$, h_n can be represented by a CFLOBDD of size $\mathcal{O}(n)$. In contrast, the size of a TIDD representation of h_n (with the same variable ordering) is $\Omega(2^n)$.

Proof. CFLOBDD Claim. We will use the variable ordering $(x_0 \ldots x_{n^2-1})$. A CFLOBDD representing h_n has n^2 variables and $2\log(n)+1$ levels: $0, \ldots, 2\log(n)$. Every proto-CFLOBDD at level-$\log(n)$ encodes a function over n variables.

The CFLOBDD of interest has groupings of two kinds, stratified by the level at which they appear: (a) one kind makes up levels 0 through $\log(n)$; (b) the other kind makes up levels $\log(n)+1$ through $2\log(n)$. The groupings in (a) implement the functions f_i, $0 \leq i \leq n-1$; the groupings in (b) implement the conjunction $\bigwedge_{i=0}^{n-1}$ in the definition of h_n. We start by describing the structure of the groupings in (a).

Let us create a new series of functions $F(2^k, s) = g_s(\text{over } 2^k \text{ variables})$, where $k = 1 \ldots l$, $2^l = n$, and $s = 0 \ldots 2^k - 1$, and $g_s(\text{over } 2^k \text{ variables}) = 1$, if the sth bit of the 2^k variables is 0, otherwise 1. We can observe that, for $2^k = n$,

g_s(over n variables) $= f_s(X_s)$. All 2^{2^k} assignments of length 2^k can be split into two buckets: $F(2^k, s) = 1$ and $F(2^k, s) = 0$. The functions in this series over 2^k variables would be: $F(2^k, s = 0)$, $F(2^k, s = 1)$, ..., $F(2^k, s = 2^k - 1)$.

Let us look at some functions in this series:

1. $F(2, 0)$ splits the assignments of length 2 into $[\{00, 10\}, \{01, 11\}]$. The first set consists of the 2-bit strings with a 0 in the 0th position. The proto-CFLOBDD is shown in Fig. 6.8a.

2. $F(2, 1)$ splits the assignments of length 2 into $[\{00, 01\}, \{10, 11\}]$. The first set consists of the 2-bit strings with a 0 in the 1st position. The proto-CFLOBDD is shown in Fig. 6.8b.

The next functions in this series would be:

1. $F(4, 0)$ splits the assignments of length 4 into

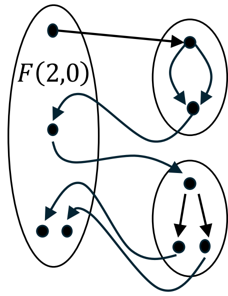
   $$[\{0000, 0010, 0100, 0110, \ldots, 1110\}, \{0001, 0011, 0101, 0111, \ldots, 1111\}]$$

   The first set consists of the 4-bit strings with a 0 in the 0th position. The proto-CFLOBDD, shown in Fig. 6.8c, recursively calls a `NoDistinctionNode` for the AConnection and the proto-CFLOBDD for $F(2, 1)$ for the BConnection.

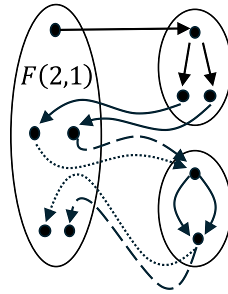
2. $F(4, 1)$ splits the assignments of length 4 into

   $$\left[\begin{array}{l} \{0000, 0001, 0100, 0101, \ldots, 1100, 1101\}, \\ \{0010, 0011, 0110, 0111, \ldots, 1110, 1111\} \end{array}\right]$$

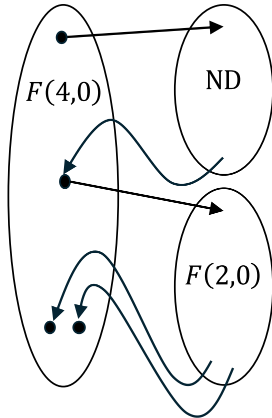
   The first set consists of the 4-bit strings with a 0 in the 1st position. Similarly, the proto-CFLOBDD, shown in Fig. 6.8d, recursively calls a `NoDistinctionNode` for the AConnection and the proto-CFLOBDD for $F(2, 2)$ for the BConnection.



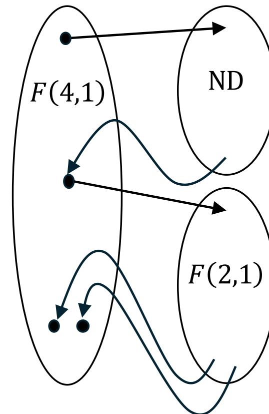
(a) proto-CFLOBDD for $F(2,0)$



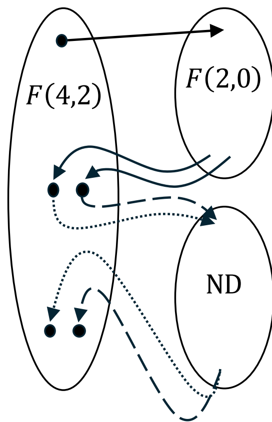
(b) proto-CFLOBDD for $F(2,1)$



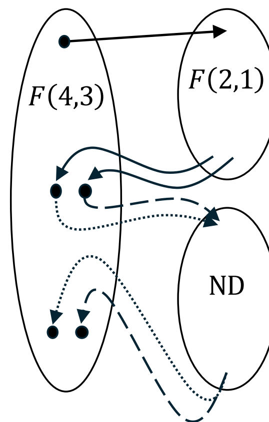
(c) proto-CFLOBDD for $F(4,0)$, recursively calls $F(2,0)$



(d) proto-CFLOBDD for $F(4,1)$, recursively calls $F(2,1)$

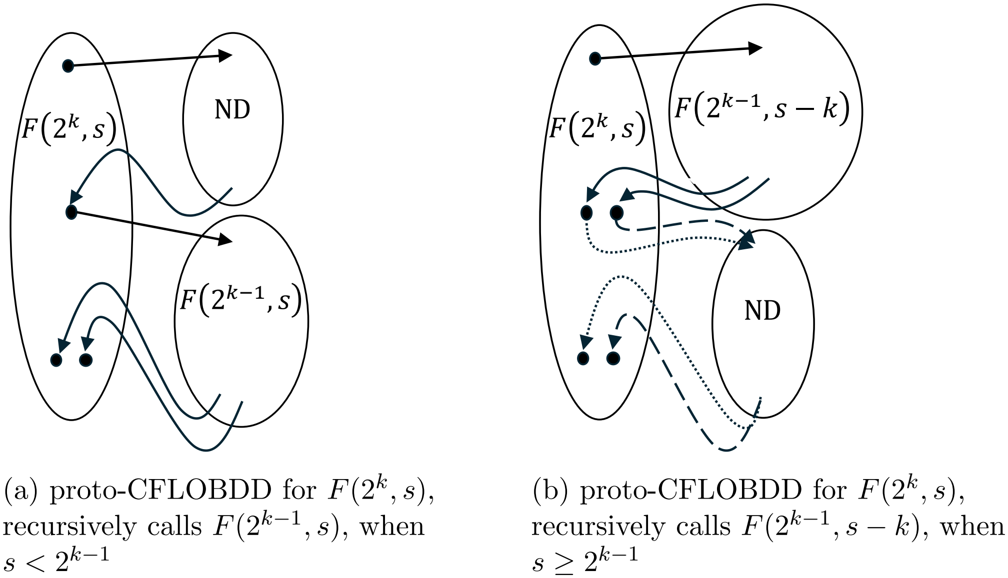


(e) proto-CFLOBDD for $F(4,2)$, recursively calls $F(2,0)$



(f) proto-CFLOBDD for $F(4,3)$, recursively calls $F(2,1)$

Figure 6.8: The diagrams show proto-CFLOBDDs for $F(2,.)$, $F(4,.)$. `ND` represents `NoDistinctionNode`.



(a) proto-CFLOBDD for $F(2^k, s)$, recursively calls $F(2^{k-1}, s)$, when $s < 2^{k-1}$

(b) proto-CFLOBDD for $F(2^k, s)$, recursively calls $F(2^{k-1}, s-k)$, when $s \geq 2^{k-1}$

Figure 6.9: Diagrams show proto-CFLOBDDs for $F(k, s)$ when $s < 2^{k-1}$ and $s \geq 2^{k-1}$. `ND` represents `NoDistinctionNode`.

3. $F(4, 2)$ splits the assignments of length 4 into

$$\begin{bmatrix} \{0000, 0001, 0010, 0011, 1000, 1001, 1010, 1011\}, \\ \{0100, 0101, 0110, 0111, 1100, 1101, 1110, 1111\} \end{bmatrix}$$

   The first set consists of the 4-bit strings with a 0 in the 2nd position. The proto-CFLOBDD, shown in Fig. 6.8e, recursively calls a proto-CFLOBDD for $F(2, 1)$ for the AConnection and `NoDistinctionNodes` for the BConnections.

4. $F(4, 3)$ splits the assignments of length 4 into

$$\begin{bmatrix} \{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111\}, \\ \{1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\} \end{bmatrix}$$

   The first set consists of the 4-bit strings with a 0 in the 3rd position. The proto-CFLOBDD, shown in Fig. 6.8f, recursively calls a proto-CFLOBDD for $F(2, 2)$ for the AConnection and `NoDistinctionNodes` for the BConnections.

If $s < 2^{k-1}$, then a proto-CFLOBDD for $F(2^k, s)$ has a `NoDistinctionNode` for AConnection and the proto-CFLOBDD for $F(2^{k-1}, s)$ as the BConnection. Similarly,

if $s \geq 2^{k-1}$, then the proto-CFLOBDD for $F(2^k, s)$ has the proto-CFLOBDD for $F(2^{k-1}, s-k)$ as the AConnection and `NodeDistinctionNodes` as BConnections.

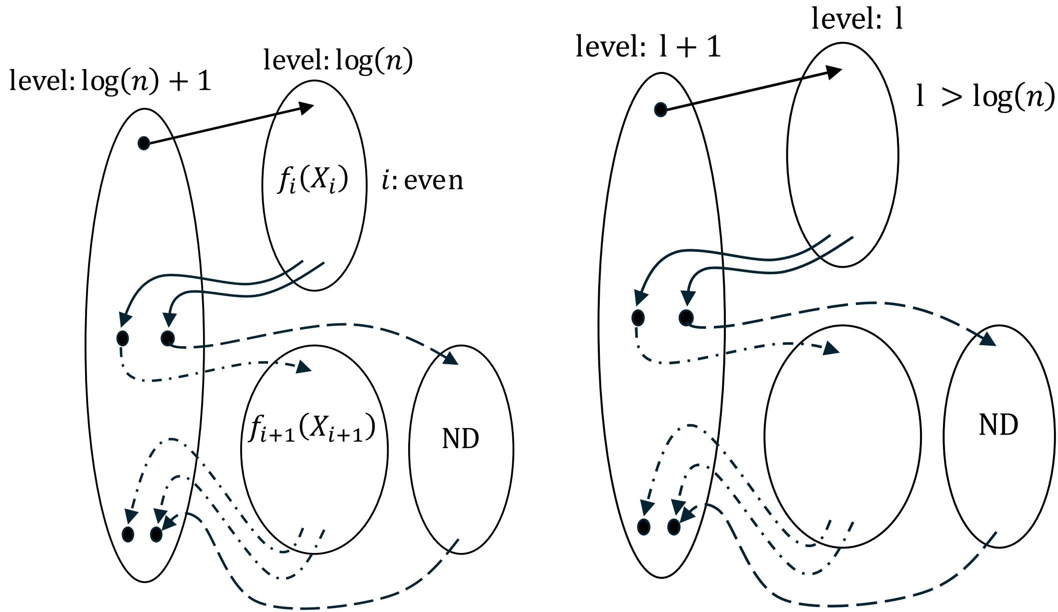
Using this information, we can make the following observations about the sizes of these proto-CFLOBDDs. Let $\mathbb{S}(F(2^k, s))$ and $\mathbb{E}(F(2^k, s))$ represent the number of unique groupings and edges of the proto-CFLOBDD representing $F(2^k, s)$, respectively, of the proto-CFLOBDD used to represent $F(2^k, s)$.

$$\begin{array}{rcr|rcr} \mathbb{S}(F(2,0)) & = & 3 & \mathbb{E}(F(2,0)) & = & 5 \\ \mathbb{S}(F(2,1)) & = & 3 & \mathbb{E}(F(2,1)) & = & 7 \\ \mathbb{S}(F(4,0)) & = & 5 & \mathbb{E}(F(4,0)) & = & 14 \\ \mathbb{S}(F(4,1)) & = & 5 & \mathbb{E}(F(4,1)) & = & 16 \\ \mathbb{S}(F(4,2)) & = & 5 & \mathbb{E}(F(4,2)) & = & 16 \\ \mathbb{S}(F(4,3)) & = & 5 & \mathbb{E}(F(4,3)) & = & 18 \end{array}$$

For $F(4,0)$, the number of groupings is equal to the number of groupings of $F(2,0)$ + 1 (for `NoDistinctionNode`) + 1 (for grouping at level-3) = 5, and the number of edges is the number of edges of $F(2,0)$ + 4 (for `NoDistinctionNode`) + 4 (for edges between level-2 and level-3 groupings) = 14. Similarly, we can count the number of groupings for others as well.

Extending this argument to general values of k and s, the number of groupings of $F(k, s)$ is $\mathbb{S}(F(2^k, s)) = \mathbb{S}(F(2^{k-1}, s)) + 2$, and the number of edges is $\mathbb{E}(F(2^k, s)) = \mathbb{E}(F(2^{k-1}, s)) + 9$, if $s < 2^{k-1}$ and $\mathbb{S}(F(2^k, s)) = \mathbb{S}(F(2^{k-1}, s-k)) + 2$ $\mathbb{E}(F(2^k, s)) = \mathbb{E}(F(2^{k-1}, s-k)) + 11$, otherwise (see Figs. 6.9a and 6.9b). That is, with each additional increase in k, the number of groupings increases by 2, and the number of edges increases by 9-11, i.e., a constant factor.

Now, let us look at the CFLOBDD that represents h_n. Each of the groupings at level-$\log(n)$ encodes f_i where $i = 0, \ldots, n-1$. The proto-CFLOBDD representing $f_i(X_i)$ is encoding the function $F(n, i)$. So, there are n proto-CFLOBDDs corresponding to the n functions $f_i(X_i) = F(n, i)$. Each proto-CFLOBDD recursively calls a proto-CFLOBDD encoding function $F(n/2, i')$, where i' depends on the relationship with n. Because there are n $F(n, i)$ functions, consequently, there would be



(a) The diagram shows the proto-CFLOBDD structure at level-$\log(n)+1$. The groupings at level-$\log(n)+1$ have an AConnection to $f_i(X_i)$ (where i is even), and two BConnections: $f_{i+1}(X_{i+1})$ with return tuple $[1,2]$ and `NoDistinctionNode` with return tuple $[2]$.

(b) The diagram shows the proto-CFLOBDD structure at level-$l+1$, $l > \log(n)$. The groupings at level-$l+1$ have an AConnection and two BConnections as shown.

Figure 6.10: Diagrams showing proto-CFLOBDDs of h_n at level-$\log(n)$+1 (Fig. 6.10a) and level-$l+1$, $l > \log(n)+1$(Fig. 6.10b).

$n/2$ number of $F(n/2, i')$ functions altogether. So at every level-p, $p = 1, \ldots, \log(n)$, there would be 2^p proto-CFLOBDDs representing $F(2^p, j)$, where $j = 0, \ldots, 2^p - 1$. The number of groupings of all proto-CFLOBDDs encoding functions in the series $f_i(X_i) = F(n, i)$, for $i = 0, \ldots, n-1$, is $n + n/2 + n/4 + \ldots + 1 = 2n$. The number of edges is constant per grouping and hence linear in the number of groupings. The total size of all proto-CFLOBDDs encoding functions in the series $f_i(X_i) = F(n, i)$ is $\mathcal{O}(n)$.

Note that we still need to count the number of groupings and edges at levels $\log(n)$+1 through $2\log(n)$. These groupings will have the structure of a perfect binary

tree of height $\log(n)$. At every level-$l(> \log(n))$, the proto-CFLOBDD (see Fig. 6.10a) would have two exits with (1) a recursive call to a unique AConnection grouping with two return edges, (2) a unique grouping through the first BConnection with two return edges to exits: $[1, 2]$, (3) a `NoDistinctionNode` through the second BConnection with one return edge to exit: $[2]$. At every level-l, $l = \log(n) + l'$, $0 < l' \leq \log(n)$ (see Fig. 6.10b), there would be $n/2^{l'}$ groupings and a constant number of edges per grouping. So the total number of groupings and edges from level-$\log(n) + 1$ to level-$2\log(n) = n/2 + n/4 + \ldots + 1 = n - 1 = \mathcal{O}(n)$. (We used only the count of the number of groupings in the calculation, because the number of edges is constant per grouping.)

The total size of the CFLOBDD that represents h_n is $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

TIDD Claim.

Let us focus on the level-$\log(n)$ of the TIDD representation for h_n. The number of states at level-$\log(n)$ would be the partitions formed by $f_i(X_i)$. Each $f_i(X_i)$ creates a partition space over 2^n assignments, and the number of states is equal to the distinct partitions of assignments satisfying all $f_i(X_i)$ partition spaces. We will calculate this by considering each $f_i(X_i)$ at a time.

- $i = 0$: f_0 divides the assignments into $[\{0, 2, 4, 6, \ldots\}, \{1, 3, 5, 7, \ldots\}]$ (we are using the integer values of the Boolean assignments). #states of TIDD = 2.
- $i = 1$: f_1 divides the assignments into $[\{0, 1, 4, 5, \ldots\}, \{2, 3, 6, 7, \ldots\}]$. The new states would have the following $L^{\downarrow_f}$ (again, using integer values): $[\{0, 4, 8, \ldots\}, \{1, 5, 9, \ldots\}, \{2, 6, 10, \ldots\}, \{3, 7, 11, \ldots\}]$. #states = 4.
- $i = 2$: f_2 divides the assignments into $[\{0, 1, 2, 3, \ldots\}, \{4, 5, 6, 7, \ldots\}]$. The new states would have the following $L^{\downarrow_f}$ (again, using integer values): $[\{0, 8, \ldots\}, \{1, 9, \ldots\}, \{2, 10, \ldots\}, \{3, 11, \ldots\}, \ldots, \{7, 15, \ldots\}]$. #states = 8.
- $\ldots$

- $i = s$: the new states would partition the space of 2^n strings into 2^{s+1} partitions, where the jth partition has elements that satisfy $e\%2^{s+1} = j$, where e is an element in the jth partition. Hence, #states = 2^{s+1}.

Hence, for $i = n-1$, i.e., the total number of states at level-$\log n$ of the TIDD, considering all f_i functions, is $2^{n+1} = \Omega(2^n)$. Therefore, the total number of states of the TIDD that represents h_n is $\Omega(2^n)$.

□

This example illustrates how the linear structure of CFLOBDDs enables an exponential compression compared to TIDDs under the same variable ordering. In a CFLOBDD, each subfunction $f_i(X_i)$ can be represented independently by a proto-CFLOBDD, and the proto-CFLOBDD for $f_{i+1}(X_{i+1})$ is connected directly to the output of $f_i(X_i)$—see Fig. 6.10a. In contrast, TIDDs lack this linear compositionality: the same set of states must be reused to represent all subfunctions $f_{i+1}(X_{i+1})$, which forces the representation to encode exponentially many distinctions and thus leads to an exponential blow-up in the number of states.

6.5 Evaluation

In this section, we provide some empirical evidence of how CFLOBDDs perform better than TIDDs in a practical domain, thereby showing that the linear structure in CFLOBDDs helps in more efficient representations than TIDDs in case of representing Boolean functions. We use the domain of quantum simulation—in particular, the Greenberger–Horne–Zeilinger (GHZ), Bernstein–Vazirani (BV), and Deutsch–Jozsa (DJ) algorithms where CFLOBDDs were shown to perform exceptionally better than BDDs. We ran our experiments on on a system running Ubuntu with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, 64 vCPUs, and 16GB of memory. Each experiment was executed five times for increasing numbers of qubits, and we report the average results across these runs.

Benchmark	#Qubits	CFLOBDD					TIDD				
		#Groupings	#Edges	#Total	Time(s)	Max. Size	#Nodes	#Edges	#Total	Time(s)	Max. Size
GHZ	131072	139	831	970	**14.87**	970	19	213	**232**	35.22	**233**
	262144	147	879	1026	**36.21**	1026	20	225	**245**	71.16	**246**
	524288	155	927	1082	**97.25**	1082	21	237	**258**	144	**259**
	1048576	163	975	1138	**228.21**	1138	22	249	**271**	292.9	**272**
	2097152	Timeout (15min)					23	261	**284**	**609.81**	**285**
	4194304	Timeout (15min)					Timeout (15min)				
BV	16	30	174	204	**0.003**	**242**	7	143	**150**	0.012	321
	32	40	237	277	**0.004**	**324**	8	240	**248**	0.403	832
	64	54	323	**377**	**0.005**	**434**	9	449	458	200.12	37731
	128	77	456	**533**	**0.007**	**617**	Timeout (15min)				134480
DJ	512	33	170	**203**	**0.004**	**290**	12	202	214	1.27	21025
	1024	36	186	**222**	**0.006**	**318**	13	222	235	10.37	65924
	2048	39	202	**241**	**0.009**	**346**	14	242	256	75.46	200207
	4096	42	218	**260**	**0.012**	**374**	15	262	277	659.62	570940
	8192	45	234	**279**	**0.024**	**402**	Timeout (15min)				

Table 6.1: Performance of CFLOBDDs and TIDDs on quantum benchmarks—*GHZ*, *BV*, *DJ*.

Tab. 6.1 reports the performance of CFLOBDDs and TIDDs on three quantum benchmarks—*GHZ*, *BV*, and *DJ*, chosen to illustrate the effect of the absence of linearity in TIDDs. We measure both the construction time and the size of the data structures representing the final quantum state for each benchmark. Tab. 6.1 also shows the maximum size of the intermediate state vectors or (unitary) matrices representation gates created when running the benchmark, which has a huge impact on the overall performance of TIDDs and CFLOBDDs.

The size of a TIDD is computed as the sum of (i) the number of internal and leaf nodes (illustrated in Fig. 6.6) and (ii) the number of edges in the transition lists. We note that the states are not explicitly stored; consequently, there is exactly one node per layer, and the total number of nodes equals the number of levels in the TIDD.

Furthermore, each transition list is represented as a list of lists. These lists are hash-consed, and each distinct list is counted only once when computing the overall size.

For the *GHZ* benchmark, as shown in Tab. 6.1, we observe that TIDDs and CFLOBDDs perform comparably, with TIDDs exhibiting slightly better performance. This is attributable to the structure of the benchmark, which consists of a sequence

of simple matrix-multiplication operations, resulting in relatively small intermediate states and gates.

In contrast, for the *BV* and *DJ* benchmarks, CFLOBDDs significantly outperform TIDDs in both time and space. Although the sizes of the final state vectors represented by CFLOBDDs and TIDDs are similar, the maximum sizes of the intermediate states and gates[4] differ substantially. In particular, for TIDDs, the intermediate representations can grow to as many as 570K nodes and edges, whereas CFLOBDDs can represent both intermediate states and gates much more compactly, which directly affects the time taken to run the benchmark.

This behavior arises because CFLOBDDs can decompose a function into smaller sub-functions, represent each sub-function efficiently, and then compose them using the inherent linear structure of CFLOBDDs—a capability that is not available in TIDDs. Consequently, these results demonstrate that, even in practice, the linear structure of CFLOBDDs plays a crucial role in enabling significantly more compact function representations.

[4] In the case of the *BV* benchmark for 128 qubits, we report the maximum size observed so far before TIDDs timeout.

Chapter 7: GCFLOBDDs: Generalized Context-Free-Language Ordered Binary Decision Diagrams

7.1 Introduction

As discussed in Chapter 3, CFLOBDDs follow a matched-path principle with a restricted structure of dividing the variables into two halves at every level, with each half belonging to either an "AConnection" or a "BConnection".

In other words, CFLOBDDs are based on the following decomposition pattern (or "pattern of calls"):

$$\begin{array}{lcl} matched_k & \rightarrow & matched_{k-1} \quad matched_{k-1} \\ matched_0 & \rightarrow & \epsilon \end{array}$$

Naturally, one can think of other decomposition schemes; for instance, in §3.7.3, it would have been useful to be able to define the ADD_n relation using the decomposition

$$\begin{array}{lcl} matched_k & \rightarrow & matched_{k-1} \quad matched_{k-1} \quad matched_{k-1} \\ matched_0 & \rightarrow & \epsilon \end{array}$$

which would have avoided having to "waste" one-quarter of the variables, as in Figs. 3.14 and 3.15.

Another possibility would be to permit a level-i grouping to have connections to level-i-j groupings, where $j \geq 1$). For instance, one could have a Fibonacci-like decomposition

$$\begin{array}{lcl} matched_k & \rightarrow & matched_{k-1} \quad matched_{k-2} \\ matched_1 & \rightarrow & \epsilon \\ matched_0 & \rightarrow & \epsilon \end{array}$$

One can also explore the possibility of having small BDDs at lower levels and the procedure-call mechanism at higher levels; this could also remove the overhead of extra structural constraints that need to be maintained by CFLOBDDs at lower levels.

As long as there is a fixed structural-decomposition pattern to how the levels connect, it should still be possible to keep the same properties—i.e., the ability to perform operations implicitly, without having to instantiate the decision tree; canonicalness; etc.—as enjoyed by CFLOBDDs.

In this chapter, we introduce *Generalized Context-Free-Language Ordered Binary Decision Diagrams* (GCFLOBDDs). GCFLOBDDs are an extension of CFLOBDDs but without a restricted and a default balanced grammar structure. They enjoy all the good properties of CFLOBDDs, and can also lead to better representations for certain functions. Because GCFLOBDDs also support BDDs internally, they can also leverage the simplicity of BDDs for representing functions or sub-functions that when represented by CFLOBDDs can lead to more complex structures.

Note that, one can always simulate a GCFLOBDD using a CFLOBDD with additional "variable padding" by ensuring that the "local relative structure and positions" of the variables in GCFLOBDD are obeyed in the padded-CFLOBDD. However, as a user of CFLOBDDs, it might become tedious for one to come up with and keep track of the appropriate padding information for every application. GCFLOBDDs can alleviate this problem by using a custom grammar or decomposition for a particular application without any additional necessary variable paddings.

7.2 Generalized CFLOBDDs (GCFLOBDDs)

The structure of GCFLOBDDs is similar to that of CFLOBDDs, except that the user specifies the decomposition pattern. This decomposition pattern is provided as a grammar.[1] We will formally define the grammar in §7.2.1 and the structure of GCFLOBDDs in §7.2.2.

[1] It could be provided in various ways, such as a derivation tree, etc. We chose a grammar representation.

7.2.1 Grammar

The grammar provided to a GCFLOBDD consists of two key characteristics: (i) the grammar is not recursive, and (ii) each nonterminal is on the left-hand side of exactly one production. Formally, the grammar can be defined as $\mathcal{G} = \langle \mathcal{N} \cup \{a\} \cup \{\mathrm{BDD}(.)\}, \mathcal{S}, \gamma \rangle$; where

- $\mathcal{N}$ represents the set of non-terminals.
- $\{a\}$ represents a single terminal;
- BDD(.) is a special non-terminal, which takes an integer argument. Henceforth, we assume that a set of non-terminals $\{\mathrm{BDD}(.)\}$ (with appropriate integer arguments) is always included in a grammar's non-terminals, and will usually not list them explicitly when a grammar is specified.
- $\mathcal{S}$ indicates the start symbol.
- γ indicates the productions.

The grammar $\mathcal{G}$ must satisfy the following constraints:

- Every non-terminal is associated with a number. For example, S_k, A_i, B_j where k, i, j are numbers.
- The number associated with a non-terminal is called the "level" of the non-terminal.
- Each non-terminal is on the left-hand side of exactly one production. That is, there cannot be two productions of the form $S_k \to S_i A_j$ and $S_k \to B_j A_l$.
- Every production has either (i) zero, or (ii) two or more non-terminals on the right-hand side.

- Every production $S_k \rightarrow A_{i_1}...A_{i_m}$ is "strictly level decreasing": (i) there is at least one non-terminal A_{i_j} such that $i_j = k - 1$ and (ii) for all i_l such that $1 \leq l \leq m$, $i_l < k$.[2]

- There is at most one level-0 non-terminal S_0 whose production is $S_0 \rightarrow a$.

- For each value of n, there is at most one level-0 non-terminal T_0 whose production is $T_0 \rightarrow \text{BDD}(n)$.

The language $L(\mathcal{G})$, which controls the structure of a $\mathcal{G}$-GCFLOBDD, consists of just a single word of the form a^k, for some k.[3] However, because each a corresponds to a Boolean variable, which can have the value 0 or 1, each $\mathcal{G}$-GCFLOBDD has 2^k matched paths (see §7.2.2). Each matched path has an associated terminal value; when the terminal values are Booleans, there can be 2^k different $\mathcal{G}$-GCFLOBDDs.

Examples of some grammars include:

Example 7.1. Example of a balanced grammar with 4 levels.

$$\mathcal{G} = \langle \{S_0, S_1, S_2, S_3, S_4, a\}, S_4, \begin{array}{ccc} S_4 & \rightarrow & S_3 S_3 \\ S_3 & \rightarrow & S_2 S_2 \\ S_2 & \rightarrow & S_1 S_1 \\ S_1 & \rightarrow & S_0 S_0 \\ S_0 & \rightarrow & a \end{array} \rangle$$

Example 7.2. Example of a ternary grammar with 4 levels.

$$\mathcal{G} = \langle \{S_0, S_1, S_2, S_3, S_4, a\}, S_4, \begin{array}{ccc} S_4 & \rightarrow & S_3 S_3 S_3 \\ S_3 & \rightarrow & S_2 S_2 S_2 \\ S_2 & \rightarrow & S_1 S_1 S_1 \\ S_1 & \rightarrow & S_0 S_0 S_0 \\ S_0 & \rightarrow & a \end{array} \rangle$$

---

[2] The indices associated with a non-terminal reflect the level, i.e., for each production $S_k \rightarrow A_{i_1} \ldots A_{i_m}$, we have $k = max(i_1, \ldots i_m) + 1$.

[3] This property is similar to whole-program paths Larus (1999) where a grammar-like formalism is used to specify a single-word language.

Example 7.3. Example of a Fibonacci grammar with 4 levels.

$$\mathcal{G} = \langle \{S_0, S_1, S_2, S_3, S_4, a\}, S_4, \begin{array}{ccc} S_4 & \rightarrow & S_3 S_2 \\ S_3 & \rightarrow & S_2 S_1 \\ S_2 & \rightarrow & S_1 S_0 \\ S_1 & \rightarrow & S_0 S_0 \\ S_0 & \rightarrow & a \end{array} \rangle$$

Example 7.4. Example of a grammar with two BDD-generating non-terminals and 4 levels.

$$\mathcal{G} = \langle \{S_0, S_1, S_2, S_3, S_4, a, A_0, B_0, BDD(.)\}, S_4, \begin{array}{ccc} S_4 & \rightarrow & S_3 A_0 \\ S_3 & \rightarrow & S_2 B_0 \\ S_2 & \rightarrow & S_1 S_0 \\ S_1 & \rightarrow & S_0 S_0 \\ S_0 & \rightarrow & a \\ A_0 & \rightarrow & BDD(5) \\ B_0 & \rightarrow & BDD(3) \end{array} \rangle$$

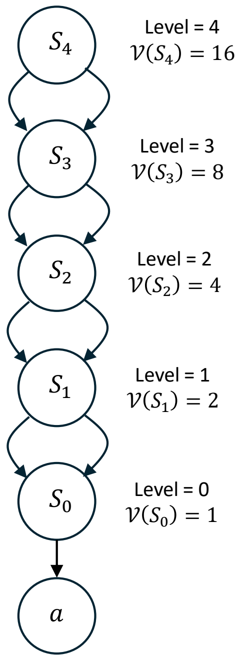
Variable Function. Let us consider a function $\mathcal{V}_\mathcal{G} : \mathcal{N} \rightarrow \mathbb{N}$, a function mapping every non-terminal of a grammar $\mathcal{G}$ to a number. This number indicates the number of variables associated with the non-terminal and is defined as follows:

$$\mathcal{V}_\mathcal{G}(S_k) = \begin{cases} \sum_{l=1}^{m} \mathcal{V}_\mathcal{G}(A_{i_m}) & S_k \rightarrow A_{i_1}...A_{i_m} \\ 1 & k = 0, S_0 \rightarrow a \\ l & k = 0, S_0 \rightarrow \text{BDD}(l) \end{cases}$$

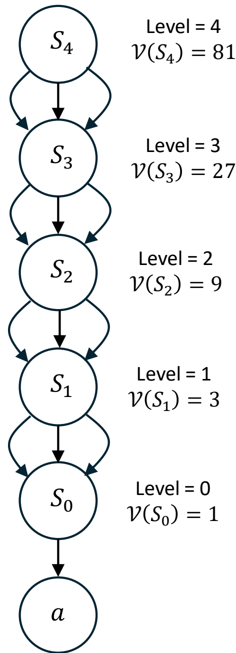
Using the above definition, we can calculate the value of the variable function for the start symbols in Exs. 7.1, 7.2, 7.3, and 7.4. For S_4 in Ex. 7.1, $\mathcal{V}(S_4) = 16$, in Ex. 7.2, $\mathcal{V}(S_4) = 81$, in Ex. 7.3, $\mathcal{V}(S_4) = 8$, and in Ex. 7.4, $\mathcal{V}(S_4) = 11$.

Level Function. Let us define $\mathcal{L}_\mathcal{G} : \mathcal{N} \rightarrow \mathbb{N}$, to compute the "level" of the non-terminal of a grammar $\mathcal{G}$.

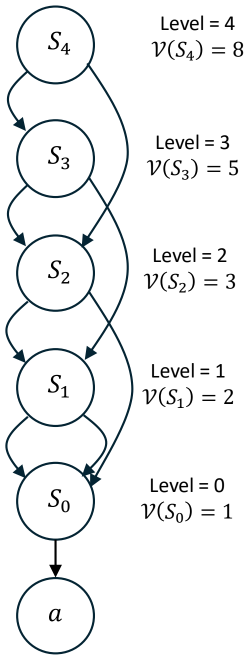
$$\mathcal{L}_\mathcal{G}(S_k) = \begin{cases} 1 + \max_{l=1}^{m} \mathcal{L}_\mathcal{G}(A_l) & S_k \rightarrow A_{i_1}...A_{i_m} \\ 0 & k = 0, S_0 \rightarrow a \\ 0 & k = 0, S_0 \rightarrow \text{BDD}(l) \end{cases}$$



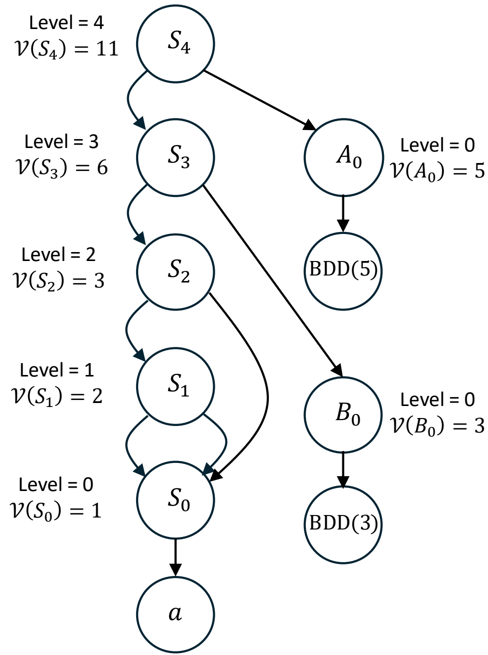
(a) DAG of grammar in Ex. 7.1.



(b) DAG of grammar in Ex. 7.2.



(c) DAG of grammar in Ex. 7.3.



(d) DAG of grammar in Ex. 7.4.

Figure 7.1: DAG representation of grammars in Exs. 7.1, 7.2, 7.3, and 7.4.

Implementation Details. In our implementation, a user-provided grammar is converted into a DAG, which is later traversed when constructing and operating on GCFLOBDDs. Fig. 7.1 show the DAG representation of the grammars discussed in Exs. 7.1, 7.2, 7.3, and 7.4.

We will also define two functions, which will be later used in the algorithms for operations on GCFLOBDDs (see §7.3). Let us consider a production of the form $S_k \rightarrow A_{i_1}...A_{i_m}$, where we call the right-hand side non-terminals the "body" of the production and each A_i a child of S_k (inspired from the DAG representation). S_k is called the "head" of the production and the parent of each A_i.

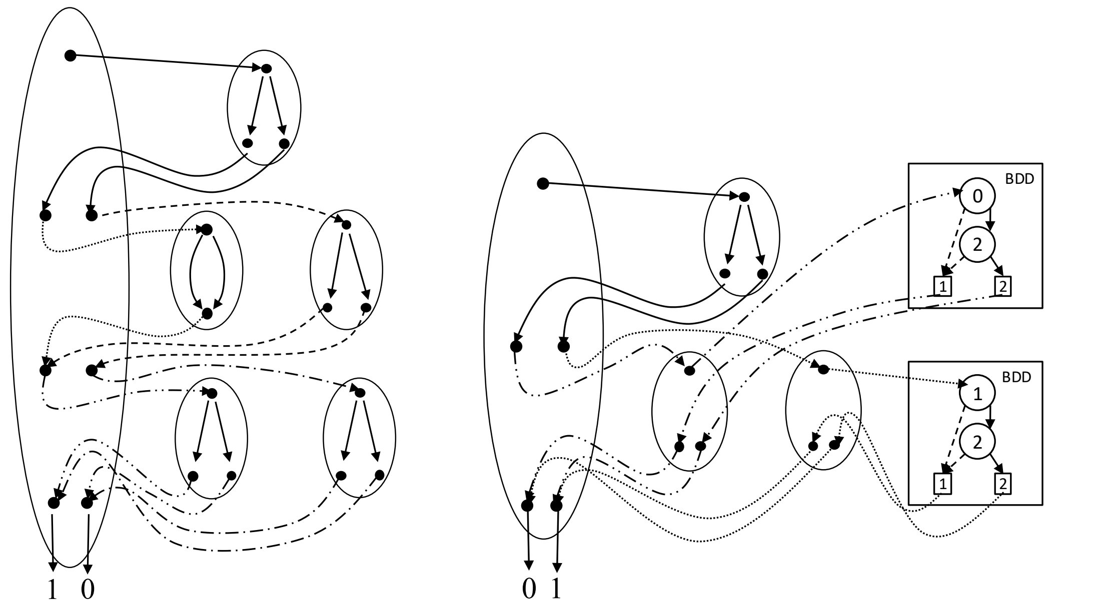
Number of Children. We define $\texttt{NumChildren}_{\mathcal{G}} : \mathcal{N} \rightarrow \mathbb{N}$, to be the function that returns the number of children of a non-terminal of grammar $\mathcal{G}$. For example, $\texttt{NumChildren}_{\mathcal{G}}(S_k) = 3, \forall k \geq 1$ in Ex. 7.2 and $\texttt{NumChildren}_{\mathcal{G}}(S_k) = 2, \forall k \geq 1$ in Exs. 7.1, 7.3, and 7.4.

Child. We define $\texttt{Child}_{\mathcal{G}} : \mathcal{N} \times \mathbb{N} \rightarrow \mathcal{N}$, to be the function that returns the ith child of a non-terminal S of grammar $\mathcal{G}$, and $0 \leq i < \texttt{NumChildren}_{\mathcal{G}}(S)$. For example, $\texttt{Child}_{\mathcal{G}}(S_3, 1) = S_2$ in Ex. 7.2 and $\texttt{Child}_{\mathcal{G}}(S_4, 1) = A_0$ in Ex. 7.4.

7.2.2 Basic Structure

Fig. 7.2 shows examples of two GCFLOBDDs. It will be useful to refer to Fig. 7.2a when reading Defn. 7.1.

Definition 7.1 Mock-GCFLOBDD**.** A *mock-GCFLOBDD* at level k is a hierarchical structure made up of some number of *groupings*, of which there is one grouping at level k, called the *head-grouping*, and at least one grouping at each level $0, 1, \ldots, k-1$. A grouping is a collection of vertices and edges (to other groupings), with the structure described below. The grouping at level k is the *head* of the mock-GCFLOBDD (conceptually "topmost;" portrayed leftmost).



(a) Function $f := (z = x \wedge y)$, variable ordering $\langle x, y, z \rangle$, grammar $\langle \{S_1, S_0, a\}, S_1, \{S_1 \rightarrow S_0S_0S_0, S_0 \rightarrow a\} \rangle$

(b) Function $f := (\bar{x_0} \wedge x_1 \wedge x_3) \vee (x_0 \wedge x_2 \wedge x_3)$, variable ordering $\langle x_0, x_1, x_2, x_3 \rangle$, grammar $\langle \{S_1, S_0, A_0, a, \text{BDD}(.)\}, S_1, \{S_1 \rightarrow S_0A_0, S_0 \rightarrow a, A_0 \rightarrow \text{BDD}(3)\} \rangle$ The `ForkGrouping` in the top-half of the figure corresponds to variable x_0, and both BDDs encode functions over the variables x_1, x_2, x_3. The node numbers of the BDDs map to the variables as $[0 \mapsto x_1, 1 \mapsto x_2, 2 \mapsto x_3]$. The root node of the BDD at lower-right is labeled 1 because of the standard ply-skipping convention in BDDs. The terminal values of such a BDD are always a sequence of the form $[1, \ldots, m]$, where m is the number of terminal nodes, and represent the (one-to-one and onto) mapping from the BDD's terminal nodes to the exit vertices of the calling level-0 grouping. The nodes among the collection of BDDs are shared, as discussed in Brace et al. (1991).

Figure 7.2: GCFLOBDD representations for two functions with different grammars.

- *Groupings:* Each oval-shaped object is called a *grouping*, and every grouping is associated with a level l (≥ 0). At least one grouping at level i is always connected to one or more groupings at level i-1.

- *Layers:* Each Grouping is divided into a set of layers ly (≥ 2): $[1 \cdots ly]$ of vertices (defined below). Different groupings can have a different number of layers.

- *Vertices:* Each layer of a Grouping g (with ly number of layers) has a set of vertices. Layer 1 has only one unique *entry vertex* (the dot at the top of g). Each of the layers 2 to $ly - 1$ has a set of vertices, called *middle vertices*; and the set of vertices in the ly^{th} layer is called the *exit vertices*. (See the dots in the middle and at the bottom of g, respectively.) The total number of vertex sets is 1 (entry vertex set) + 1 (exit vertices' set) + m (middle vertices' set), which is equal to the number of layers ly. All sets of vertices are disjoint. (It is useful to think of the collections of middle vertices and exit vertices as two sequences, each numbered from left-to-right.)

- *Connections:* The connections from a layer ly_i of a grouping g that "call" another grouping g' are called Connection-edges, and the edges from g' exit vertices to g's layer ly_{i+1} are called *return-edges*. The grouping g' is one of the Connections of g. If a grouping has ly layers, then the grouping would be split into $ly - 1$ Connection sets numbered: $\text{Connection}_1, \ldots, \text{Connection}_{ly-1}$.

  There is only one Connection at layer 1, i.e., from g's entry vertex, but (in general) there can be more than one Connection (each with a set of return edges back to g's exit vertices) at other layers. The last layer – corresponding to the grouping's exit vertices – doesn't have any connections.

- *Value Edges:* The edges that connect the exit vertices of the head-grouping to the terminal values are called *value edges*.

The groupings at level-0 (conceptually "bottommost;" portrayed to the right) are of three types:

- `ForkGrouping`: Has two layers: layer 0: unique entry vertex, layer 1: two exit vertices, and two edges (the 0-edge and 1-edge). This grouping is associated with only one variable.
- `DontCareGrouping`: Has two layers: layer 0: unique entry vertex, layer 1: one exit vertex, and two edges (the 0-edge and 1-edge), both leading to the same exit vertex. This grouping is associated with only one variable.
- `BDDGrouping`$[k]$: Has two layers: layer 0: unique entry vertex, layer 1: exit vertices (one or more), 1 Connection-edge, and one or more return-edges. This grouping is connected to a BDD encoding a function over k variables.

Relation with a grammar. Every GCFLOBDD C is parameterized by a grammar $\mathcal{G}$. Every grouping of C is associated with a grammar production rule.

- If a production is of the form $\gamma : S_k \rightarrow A^1_{i_1}...A^n_{i_m}$,[4] then the grouping g associated with γ would have $n+1$ layers. That is, the number of layers of g is one more than the number of non-terminals on the right-hand side of γ. The level of grouping g is equal to k.
- If the production is of the form $\gamma : S_0 \rightarrow a$, then g is at level-0 and is either a `DontCareGrouping` or a `ForkGrouping`.
- If the production is of the form $\gamma : S_0 \rightarrow \text{BDD}(.)$, then g is at level-0 and is a `BDDGrouping` with the entry vertex connecting to a BDD.

Fig. 7.2a shows a GCFLOBDD C_1 for the function $f := (z = x \wedge y)$ with variable ordering $\langle x, y, z \rangle$, and grammar $\langle \{S_1, S_0, a\}, S_1, \{S_1 \rightarrow S_0S_0S_0, S_0 \rightarrow a\} \rangle$. C_1 has three groupings: a grouping g at level-1 with 4 layers (associated with the production $S_1 \rightarrow S_0S_0S_0$, a `ForkGrouping` and a `DontCareGrouping`. Here, the

[4]the superscript j in $A^j_{i_h}$ indicates the jth non-terminal.

connections at layers 0, 1, and 2 encode variables x, y, and z, respectively. So, the grouping g at level-1 encodes total three variables – corresponding to the three Connection sets – one at every layer.

Fig. 7.2b shows a GCFLOBDD C_2 for the function $f := (\bar{x_0} \wedge x_1 \wedge x_3) \vee (x_0 \wedge x_2 \wedge x_3)$ with variable ordering $\langle x_0, x_1, x_2, x_3 \rangle$, and grammar $\langle \{S_1, S_0, A_0 a, \text{BDD}(.)\}, S_1, \{S_1 \rightarrow S_0 A_0, S_0 \rightarrow a, A_0 \rightarrow \text{BDD}(3)\} \rangle$. $C_2 = 2$ has four groupings: a grouping g at level-1 with 3 layers (associated with the production $S_1 \rightarrow S_0 A_0$, a `ForkGrouping` at Connection$_0$ of g and two `BDDGroupings` at Connection$_1$ of g corresponding to the production $A_0 \rightarrow \text{BDD}(.)$. The Connection at layer 1 of g encodes variable x_0. Let us refer to the two Connections at layer 2 as Connection$_{21}$ and Connection$_{22}$. Connection$_{21}$ and Connection$_{22}$ encode three variables x_1, x_2, x_3, as indicated by the production $A_0 \rightarrow \text{BDD}(.)$. Connection$_{21}$ encodes a function mapping $\{x_1, x_3\}$ to indices $[1, 2]$. and Connection$_{22}$ encodes another function mapping $\{x_2, x_3\}$ to indices $[1, 2]$. One can also view the BDDs at Connection$_{21}$ and Connection$_{22}$ as follows: the BDD [5] corresponding to Connection$_{21}$ is a function over three variables $\{0, 1\}^3 \rightarrow \mathbb{N}$. It divides the set of 2^3 strings into equivalence classes: $[1 \mapsto \{000, 001, 010, 011, 100, 110\}, 2 \mapsto \{101, 111\}]$. Similarly, the BDD corresponding to Connection$_{22}$ divides the set of 2^3 strings into the equivalence classes: $[1 \mapsto \{000, 001, 100, 101, 010, 110\}, 2 \mapsto \{011, 111\}]$. The grouping g at level-1 encodes four variables – corresponding to the two Connection sets – one variable at layer 1 and three variables at layer 2. As we can observe, although Connection$_{21}$ encodes a function over x_1, x_2, and x_3, the numbering of the BDD nodes starts with 0. This design choice supports the view that every BDD encodes a local sub-function, unaffected by the global function being encoded by the GCFLOBDD. In this case, variables x_1, x_2, and x_3 correspond to BDD nodes numbered 0, 1, and 2, respectively. The nodes among a collection of BDDs are shared as discussed in Brace et al. (1991).

[5] It is actually a Multi-Terminal BDD (MTBDD) because the terminal nodes of the BDD are integer indices. If the `BDDGrouping` encodes p variables over m exit vertices, then the corresponding BDD would have m terminal nodes: $[1, \ldots, m]$.

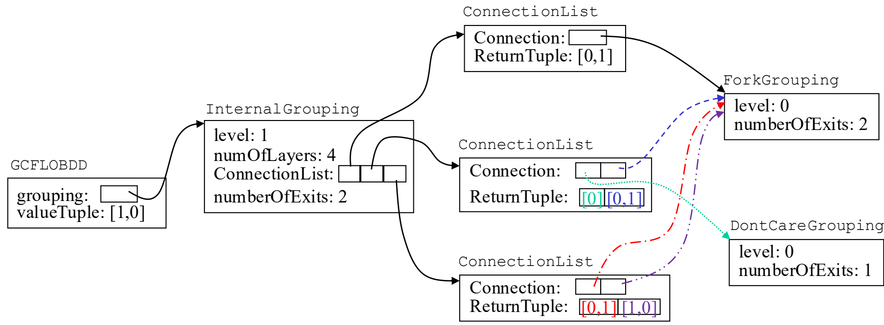


Figure 7.3: Object diagram of GCFLOBDD for the function in Fig. 7.2a.

The structure of Connection$_{22}$ is similar.

Fig. 7.3 represents an object diagram of the GCFLOBDD representation for the function in Fig. 7.2a. In this representation, there are no explicit entry, middle, and exit vertices. In the `InternalGrouping` object, the pointers to `ConnectionList` are layers [1..*numberOfLayers*−1], with the indices of the Connection pointers in the `ConnectionList` serving as vertices. Indices in [1..*numberOfExits*] stand for the exit vertices (The context determines whether an index is being used as a middle vertex or an exit vertex). The various Connection edges are stored as pointers to Groupings and elements of ReturnTuples. There are three grouping classes in Fig. 7.3: `InternalGrouping`, `ForkGrouping`, and `DontCareGrouping`. Let us compare Fig. 7.2a, the associated grammar, and the object representation in Fig. 7.3. The grouping at level-1 in Fig. 7.2a corresponds to the `InternalGrouping` object in Fig. 7.3 and the production $S_1 \rightarrow S_0S_0S_0$. The grouping at level-1 has 4 layers (there are 3 S_0's in the body of the production), and thus 3 "ConnectionList"s, represented via a (heap-allocated) array in the `InternalGrouping`. Each element of this array has a set of "Connection" pointers and return tuples. These correspond to the vertices in each layer shown in Fig. 7.2a. Each "Connection" pointer points to a level-0 grouping, corresponding to the production $S_0 \rightarrow a$, shown as a `ForkGrouping` or `DontCareGrouping` in Fig. 7.3. In the case of a `BDDGrouping`, a Connection would point to a `BDDGrouping`, which in turn would store a pointer to a BDD, and a return tuple whose length is equal to the number of terminal nodes of the BDD. The BDD

pointers are also shared across different `BDDGroupings`.

Definition 7.2 Mock-proto-GCFLOBDD**.** A *mock-proto-GCFLOBDD* at level i is a grouping at level i, together with the lower-level groupings to which it is connected (and the connecting edges). In other words, a mock-proto-GCFLOBDD has the following recursive structure:

- a mock-proto-GCFLOBDD at level 0 is either a fork grouping, a don't-care grouping, or a BDD-grouping.
- a mock-proto-GCFLOBDD at level i is headed by a grouping at level i whose Connection edges and associated return edges "call" some number of level-k mock-proto-GCFLOBDDs, such that $k < i$, and there is at least one grouping such that $k = i - 1$.

A mock-GCFLOBDD is a mock-proto-GCFLOBDD in which (i) the exit vertices of the mock-proto-GCFLOBDD have been associated with specific terminal values V.

GCFLOBDD also adheres to the principles underlying the structure of CFL-OBDDs, specifically the matched-path principle and the contextual-interpretation principle.

Matched-Path Principle. *When a path follows an edge that returns to level i from level $j < i$, it must follow an edge that matches the closest preceding edge from level i to level j.*

Contextual-Interpretation Principle. *A level-0 grouping is not associated with a specific Boolean variable. Instead, the variable(s) that a level-0 grouping refers to are determined by context, and depend on the number of variables encountered so far.*

For example, consider the GCFLOBDD in Fig. 7.2b, the node numbered 2 in the BDD corresponds to the 4th variable, i.e., x_3. When the BDD grouping is visited, variable x_0 is already interpreted. Hence, the node numbered 0 corresponds to the 2nd variable x_1 and therefore, the node numbered 2 corresponds to the 4th variable.

Canonicity. Like other decision diagrams like CFLOBDDs, WCFLOBDDs, etc., we ensure *canonicity*—every function has a unique representation—by imposing certain structural invariants (Wegener, 2000, p. 48), §3.3.1, §4.2.2.

Intuition for canonicity: Similar to CFLOBDDs, and WCFLOBDDs, the structural invariants are designed to ensure that—for a given order on the Boolean variables, and a *given grammar*—each Boolean function has a unique, canonical representation as a GCFLOBDD. In reading Defn. 7.3 below, it will help to keep in mind that the goal of the invariants is to force there to be a *unique* way to fold a given decision tree into a GCFLOBDD that represents the same Boolean function. The main characteristic of the folding method is that it works *greedily, left to right*, similar to that of CFLOBDDs. This directional bias shows up in structural invariants 1, 2a, and 2b.

Definition 7.3. A *(proto-)GCFLOBDD* is a mock-(proto-)GCFLOBDD in which every grouping/proto-GCFLOBDD satisfies the *structural invariants* given below.

Structural Invariants. The structural invariants mainly deal with (i) the organization of return tuples, and (ii) the terminal nodes of the BDDs. The following invariants deal with constraints on return tuples. Let c be a Connection edge with return tuple rt_c from g_j to g_i.

1. If c is a layer-1-Connection edge, then rt_c must map g_j's exit vertices 1-to-1, in order, onto g_i's layer-2 vertices.

2. If c is a layer-k-Connection edge, $k > 1$, whose source is middle vertex $m + 1$ of g_i and whose target is g_j, then rt_c must meet two conditions:

   (a) It must map g_j's exit vertices 1-to-1 (but not necessarily onto) g_i's vertices in layer-$k + 1$. It cannot map g_j's exit vertices to any vertex in layer-k', where $k' \neq k + 1$.

(b) It must "compactly extend" the set of layer-$(k+1)$ vertices in g_i defined by the return tuples for the previous m layer-k-Connections (similar to condition (1) on layer-1 Connection): Let rt_{c_1}, rt_{c_2}, …, rt_{c_m} be the return tuples for the first m layer-k-Connection edges out of g_i. Let S be the set of indices of layer-$(k+1)$ vertices of g_i that occur in return tuples rt_{c_1}, rt_{c_2}, …, rt_{c_m}, and let n be the largest value in S. (That is, n is the index of the rightmost exit vertex of g_i that is a target of any of the return tuples rt_{c_1}, rt_{c_2}, …, rt_{c_m}.) If S is empty, then let n be 0.

Now consider rt_c ($= rt_{c_{m+1}}$). Let R be the (not necessarily contiguous) sub-sequence of rt_c whose values are strictly greater than n. Let r be the size of R. Then R must be exactly the sequence $[n+1, n+2, \ldots, n+r]$.

3. A proto-GCFLOBDD can be used (i.e., pointed to by other higher-level groupings) multiple times, but a proto-GCFLOBDD never contains two separate instances of equal proto-GCFLOBDDs. (Equality is defined inductively on the hierarchical structure of groupings.)

4. For every pair of layer-k-Connection edges c and c' of grouping g_i, with associated return tuples rt_c and $rt_{c'}$, if c and c' lead to two level j proto-GCFLOBDDs, say p_j and p'_j, such that $p_j = p'_j$, then the associated return tuples must be different (i.e., $rt_c \neq rt_{c'}$). This condition means that two layer-k-Connection edges c and c' can "call" the same proto-GCFLOBDD, but the two sets of return edges rt_c and $rt_{c'}$ have to be different.

The following invariants pertain to the `BDDGroupings` and the associated BDDs. Let c be a Connection-edge from a `BDDGrouping` g to a BDD b with return tuple rt_c from b's terminal nodes to g's exit vertices.

5. rt_c must map 1-1, in order, onto g's exit vertices.

6. If there are k terminal nodes, then the terminal nodes must be numbered $[1, \ldots, k]$.

Property	CFLOBDDs	GCFLOBDDs
Grammar	Fixed and balanced grammar	Arbitrary grammar (parameterized)
Structure	Fixed with AConnection and BConnections	Varies based on input grammar. However, the extra need for "ConnectionList" can cause a slight increase in memory and time.
Variables	Encodes variable sets that are a power of 2. Can encode arbitary variable sets with padding	Supports an arbitrary number of variables provided by the grammar (without padding).
Canonicity	Canonical	Canonical
BDD	Does not support internal BDDs	Supports internal BDDs
Algorithms	Follows a half-and-half divide-and-conquer approach	Follows a divide-and-conquer approach based on the associated grammar.

Table 7.1: Comparison of GCFLOBDDs with CFLOBDDs.

By enforcing the above structural invariants, it can be proved that GCFLOBDDs are canonical, similar to CFLOBDDs. The proof of canonicity follows a similar argument to that of CFLOBDDs, i.e., for all C, $\mathit{Fold}(\mathit{Unfold}(\mathrm{C})) = \mathrm{C}$, and hence, we will skip the formal proof here.

7.2.3 Comparisions with CFLOBDDs

Tab. 7.1 shows a comparison of GCFLOBDDs with CFLOBDDs. GCFLOBDDs follow a similar structure and hold similar properties to those of CFLOBDDs. The main difference is that GCFLOBDDs are more flexible in structure because they support an arbitrary, user-supplied, non-recursive grammar and internal BDDs, and, as a result, they may be able to encode functions better than CFLOBDDs.

Algorithm 40: ConstantGCFLOBDD

Input: int k (level), Value v, Grammar $\mathcal{G} = \langle N \cup \{a\}, S, \gamma \rangle$
Output: GCFLOBDD representation of a function with $\mathcal{V}(S)$ variables and constant value v

```
1 begin
2     return
      RepresentativeGCFLOBDD(NoDistinctionProtoGCFLOBDD(k, G),
      [v]);
3 end
```

7.3 Operations on GCFLOBDDs

In this section, we will briefly discuss some basic operations using GCFLOBDDs. The algorithms for the operations on GCFLOBDDs follow a divide-and-conquer, recursive approach similar to that of CFLOBDDs. However, unlike CFLOBDDs, which perform recursive calls from a grouping g on structures with half the number of variables of g—i.e., g's AConnection and BConnections, in GCFLOBDDs the pattern of recursive calls at each grouping g' follows the production associated with g' in the underlying grammar. This section discusses the algorithms for constant, unary, and binary functions.

Pragmatics. Techniques such as hash-consing of groupings to maintain unique representations, function caching, and equality testing of proto-GCFLOBDDs follow the same pattern as that of CFLOBDDs (and WCFLOBDDs). We maintain two caches – one for GCFLOBDD groupings and another for BDD nodes.

7.3.1 Constant Functions

The constant functions $f_0(x) = \lambda x.\bar{0}$ and $f_1(x) = \lambda x.\bar{1}$ have exactly one grouping for every production rule of the grammar. Their GCFLOBDD-creation operations internally call `NoDistinctionProtoGCFLOBDD` , which when paired with value tuples [0] and [1] (or $[F]$ and $[T]$) form the GCFLOBDDs for the functions f_0

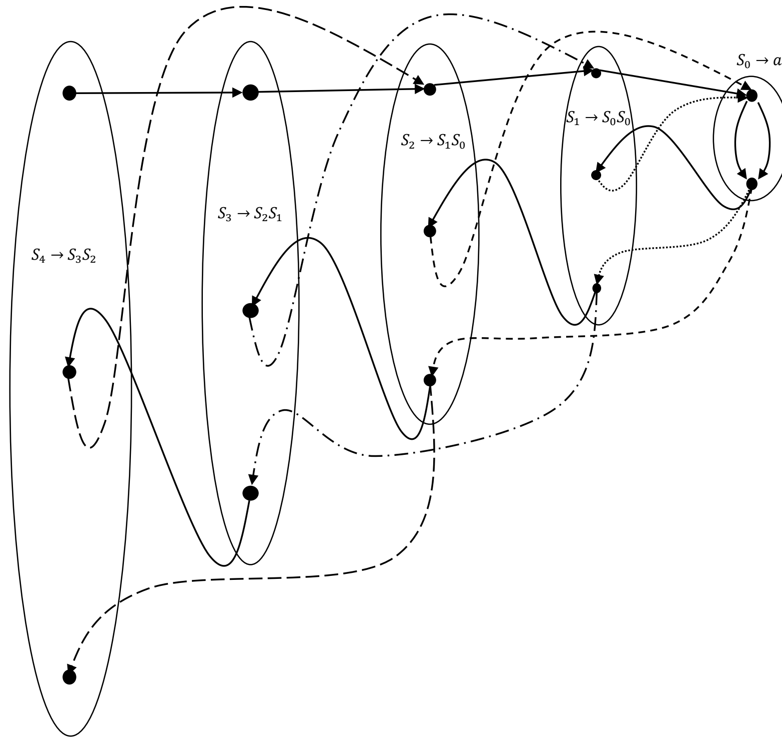


Figure 7.4: The diagram shows a `NoDistinctionProtoGCFLOBDD` for the grammar shown in Ex. 7.3. The productions associated with every grouping are also highlighted.

and f_1 respectively. Fig. 7.4 shows a `NoDistinctionProtoGCFLOBDD` for the grammar shown in Ex. 7.3. The productions associated with every grouping are also highlighted. We see that the grouping for $S_k \to S_{k-1}S_{k-2}$ has a connection at layer-0 to the `NoDistinctionProtoGCFLOBDD` corresponding to S_{k-1} followed by another `NoDistinctionProtoGCFLOBDD` for S_{k-2}. Algs. 43 and 42 show the algorithms for creating GCFLOBDDs for f_1 and f_0, respectively. The algorithms internally call `NoDistinctionProtoGCFLOBDD`, shown in Alg. 41. At level-0, the algorithm returns a `DontCareGrouping` if the production is of the form $S_0 \to a$. However, if the production leads to a BDD non-terminal, such as $T_0 \to \text{BDD}(m)$, for some $m \geq 2$, then the algorithm returns a `BDDGrouping` (`RepresentativeBDD_DontCareGrouping`) that points a BDD that encodes $\lambda x_{p_0} ... x_{p_{m-1}}.1$. However, because we are using reduced-BDDs (ROBDDs), the BDD would only have a single node representing the constant

Algorithm 41: NoDistinctionProtoGCFLOBDD

```
   Input: int k (level), Grammar 𝒢 = ⟨N ∪ {a}, S, γ⟩
   Output: Proto-GCFLOBDD representation of a function with 𝒱(S)
           variables
 1 begin
 2   if k == 0 and S → a then
 3     return RepresentativeDontCareGrouping;
 4   end
 5   if k == 0 and S → BDD(m) then
 6     return RepresentativeBDDDontCareGrouping(m);
 7   end
 8   InternalGrouping g = new InternalGrouping(ℒ(S));
 9   g.grammar = γ(S);                    // Production headed by S
10   g.numberOfLayers = NumChildren_𝒢(S) + 1;
11   for j ← 1 to g.numberOfLayers − 1 do
12     S′ = Child_𝒢(S, j);
13     Grammar 𝒢′ = ⟨N ∪ {a}, S′, γ⟩;
14     g.ConnectionList[j].Connection[1] =
        NoDistinctionProtoCFLOBDD(ℒ(S′), 𝒢′);
15     g.ConnectionList[j].ReturnTuple[1] = [1];
16   end
17   g.numberOfExits = 1;
18   return RepresentativeGrouping(g);
19 end
```

Algorithm 42: FalseGCFLOBDD

```
   Input: int k (level), Grammar 𝒢 = ⟨N ∪ {a}, S, γ⟩
   Output: CFLOBDD representation of a function with 𝒱(S) variables
           and constant value F
 1 begin
 2   return ConstantGCFLOBDD(k, F, 𝒢);
 3 end
```

function that maps all assignments to the index 1. The functions $\mathcal{V}(S)$ and $\mathcal{L}(S)$ compute the number of variables encoded by the grammar with start symbol S and the level of the production headed by S, respectively defined in §7.2.1.

Algorithm 43: TrueGCFLOBDD

Input: int k (level), Grammar $\mathcal{G} = \langle N \cup \{a\}, S, \gamma \rangle$
Output: GCFLOBDD representation of a function with $\mathcal{V}(S)$ variables and constant value T

```
1 begin
2     return ConstantGCFLOBDD(k, T, G);
3 end
```

Algorithm 44: ProjectionGCFLOBDD

```
1 Algorithm ProjectionGCFLOBDD(k, i, G)
      Input: int k (level), int i (index), Grammar G = ⟨N ∪ {a}, S, γ⟩
      Output: CFLOBDD representing function λx_0, x_1, ..., x_{n-1}.x_i
2     begin
3         assert(0 ≤ i < V_G(S));
4         return
            RepresentativeGCFLOBDD(ProjectionProtoGCFLOBDD(k,i,
            G), [F,T]);
5     end
6 end
```

7.3.2 Projection Function

A second family of GCFLOBDD-creation operations produces the Boolean-valued *(single-variable) projection functions* of the form $\lambda x_0, x_1, \ldots, x_{n-1}.x_i$, where i ranges from 0 to $n-1$. Fig. 7.5 illustrates the structure of the GCFLOBDDs that represent these functions. Fig. 7.5 shows `ProjectionGCFLOBDD` where $i > \sum_{p=1}^{p'-1} \mathcal{V}(A_{j_p})$ for $S_k \rightarrow A_{j_1}...A_{j_p}$. The first $p'-1$ layers "call" `NoDistinctionProtoGCFLOBDD`, the p'^{th} layer "calls" `ProjectionGCFLOBDD` (highlighted in blue) and the next $l - p'$ layers "call" `NoDistinctionProtoGCFLOBDD`. Algs. 44, 45, and 46 give pseudo-code for `ProjectionGCFLOBDD`$(k, i, \mathcal{G})$, which constructs the ith such function.

7.3.3 Unary Operations on CFLOBDDs

This section discusses how to perform the unary operation of scalar multiplication on GCFLOBDDs.

Algorithm 45: ProjectionProtoGCFLOBDD

```
1  SubRoutine ProjectionProtoGCFLOBDD(k, i, 𝒢)
     Input: int k (level), int i (index)
     Output: Grouping g representing function λx_0, x_1, ..., x_{2^k−1}.x_i
2    begin
3      if k == 0 and S → a then                    // i must also be 0
4        return RepresentativeForkGrouping;
5      end
6      if k == 0 and S → BDD(m) then          // for some int m > 1
7        return ProjectionBDD(m);
8      end
9      InternalGrouping g = new InternalGrouping(k);
10     g.grammar = γ(S);                       // Production headed by S
11     g.numberOfLayers = NumChildren_𝒢(S) + 1;
12     int p' = findChildIndex(𝒢, S, i);  // Find least index j_index of
         S_k → A_{j_1}...A_{j_l} such that Σ_{p=1}^{p'} 𝒱_𝒢(A_{j_v}) > i
13     for j ← 1 to p'-1 do
14       S' = Child_𝒢(S, j);
15       𝒢' = ⟨N ∪ {a}, S', γ⟩;
16       g.ConnectionList[j].Connection[1] =
           NoDistinctionProtoCFLOBDD(ℒ(S'), 𝒢');
17       g.ConnectionList[j].ReturnTuple[1] = [1];
18     end
19     S_{p'} = Child_𝒢(S, p');
20     𝒢_{p'} = ⟨N ∪ {a}, S_{p'}, γ⟩;
21     #V = Σ_{p=1}^{p'−1} 𝒱_𝒢(A_{j_v}) of S → A_{j_1}...A_{j_l};
22     i = i − #V ;  // Remove variables encoded by S's children
         before index.
23     g.ConnectionList[p'].Connection[1] =
         ProjectionProtoGCFLOBDD(ℒ(S_{p'}), i, 𝒢_{p'});
24     g.ConnectionList[p'].ReturnTuple[1] = [1,2];
```

Scalar Multiplication. Function `ScalarMultiplyGCFLOBDD` of Alg. 47 applies to any GCFLOBDD that maps Boolean-variable-to-Boolean-value assignments to values on which multiplication by a scalar value of type `Value` is defined. `ScalarMultiplyGCFLOBDD` constructs a GCFLOBDD for the constant function $\lambda x_0, x_1, \ldots, x_{n-1}.v$, which is multiplied by GCFLOBDD c using

Algorithm 46: ProjectionProtoGCFLOBDD (cont.)

```
24
25
26        for j ← p′ + 1 to g.numberOfLayers − 1 do
27            S′ = Child_G(S, j);
28            G′ = ⟨N ∪ {a}, S′, γ⟩;
29            g.ConnectionList[j].Connection[1] =
               NoDistinctionProtoCFLOBDD(L(S′), G′);
30            g.ConnectionList[j].ReturnTuple[1] = [1];
31            g.ConnectionList[j].Connection[2] =
               g.ConnectionList[j].Connection[1];
32            g.ConnectionList[j].ReturnTuple[2] = [2];
33        end
34        g.numberOfExits = 2;
35        return RepresentativeGrouping(g);
36    end
37 end
```

Algorithm 47: ScalarMultiplyGCFLOBDD

Input: GCFLOBDD c, Value v

Output: GCFLOBDD $c' = c * v$

```
1 begin
      // Multiply GCFLOBDD c by the GCFLOBDD for the constant
         function λx_0, x_1, ..., x_{n-1}.v
2     return BinaryApplyAndReduce(c, ConstantGCFLOBDD(c.level, v),
       (op)Times);                                       // (See §7.3.4)
3 end
```

`BinaryApplyAndReduce`—the generic operation for binary GCFLOBDD operations (discussed in §7.3.4)—with the multiplication operator Times passed as the third argument.

7.3.4 Binary Operations on GCFLOBDDs

This section discusses the algorithm for performing binary operations on GCFLOBDDs. Similar to CFLOBDDs, the binary operation `op` of two GCFLOBDDs $n = n_1$ `op` n_2 follows a recursive divide-and-conquer structure via a two-step process:

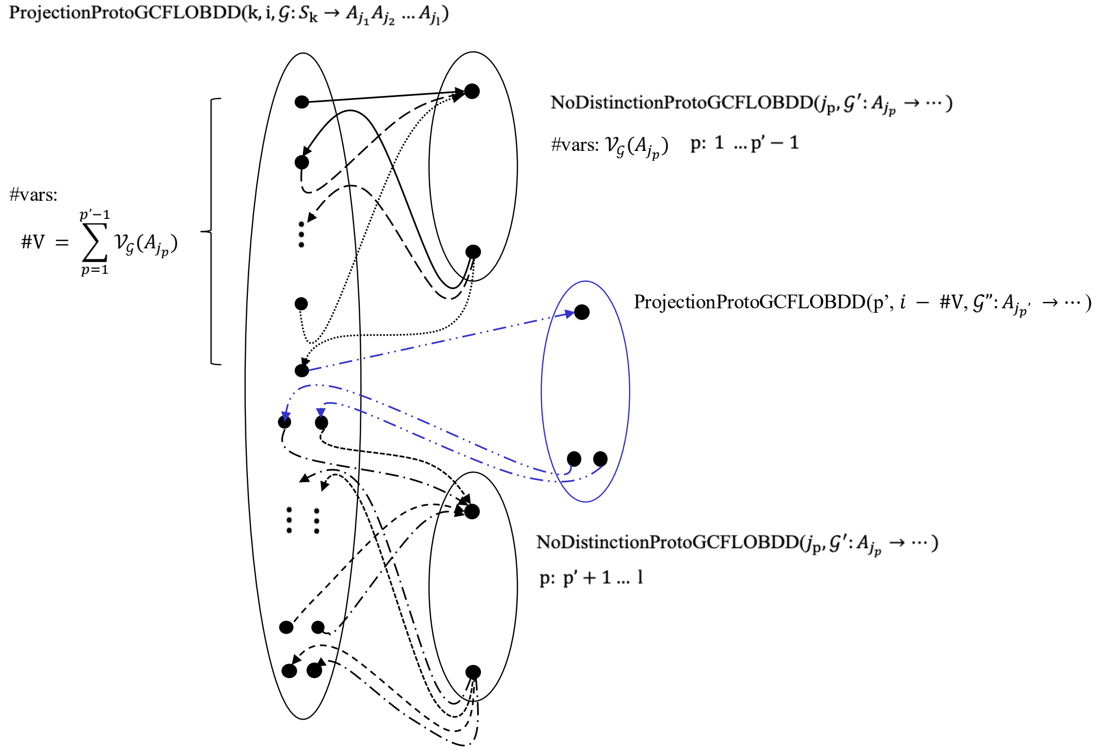


Figure 7.5: The diagram illustrates the actions taken by `ProjectionProtoGCFLOBDD`, where $i > \sum_{p=1}^{p'-1} \mathcal{V}(A_{j_p})$ for $S_k \rightarrow A_{j_1} \dots A_{j_p}$. The first $p' - 1$ layers "call" `NoDistinctionProtoGCFLOBDD`; the p'^{th} layer "calls" `ProjectionProtoGCFLOBDD` (highlighted in blue); and the next $l - p'$ layers "call" `NoDistinctionProtoGCFLOBDD`.

1. PairProduct: perform a product construction, following the grammar structure

2. Reduce: perform a reduction step on the result of step one.

The operation `op` is performed on two GCFLOBDDs n_1 and n_2 iff the grammar of both n_1 and n_2 are the same. Moreover, the grammar of the resultant n is also the same as n_1 and n_2. That is, both steps create intermediate/final GCFLOBDDs that obey the same grammar structure as that of n_1 and n_2.

As discussed in CFLOBDDs, the PairProduct step on two groupings g_1, g_2 creates list of tuples of the form $pt = [e_1, e_2]$ where e_1 and e_2 represent the exit vertices of g_1, g_2 respectively. These tuples are then used to decide the groupings

g_1', g_2' on which PairProduct is recursively called. In the case of `BDDGroupings`, which point to BDDs, the PairProduct step computes the pair product of the two BDDs and returns an intermediate BDD with a list of tuples. Each tuple refers to the terminal node numbers of the BDDs. When performing Reduce, the algorithm goes over the GCFLOBDD structure in a bottom-up fashion, recursively computing the Reduce on groupings (and BDDs) and merging the groupings (and BDDs) as indicated by the return tuple.

Fig. 7.6 shows how PairProduct and Reduce work on the two BDDs shown in Fig. 7.6a and Fig. 7.6b (which are part of larger GCFLOBDDs). Fig. 7.6c shows the result of PairProduct, leading to four terminal nodes representing the pairs $[1,1], [1,2], [2,1], [2,2]$. Fig. 7.6d shows the result of the Reduce operation on Fig. 7.6c after the four pairs are reduced by the mapping $[[1,1] \mapsto 1, [1,2] \mapsto 2, [2,1] \mapsto 2, [2,2] \mapsto 1]$.

7.4 Evaluation

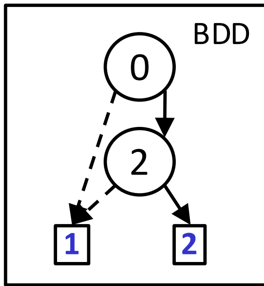
Our experiments answer two questions:

RQ1: Can GCFLOBDDs perform better than CFLOBDDs in the case of representing combinatorial circuits?

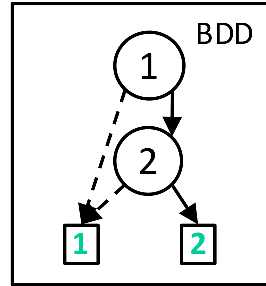
RQ2: Can GCFLOBDDs represent functions significantly better than CFLOBDDs in terms of space and time?

We ran our experiments on on a system running Ubuntu with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, 64 vCPUs, and 16GB of memory, using our implementation of CFLOBDDs (as discussed in Chapter 3[6]) and our implementation of GCFLOBDDs.[7] We measure performance of GCFLOBDDs and CFLOBDDs in

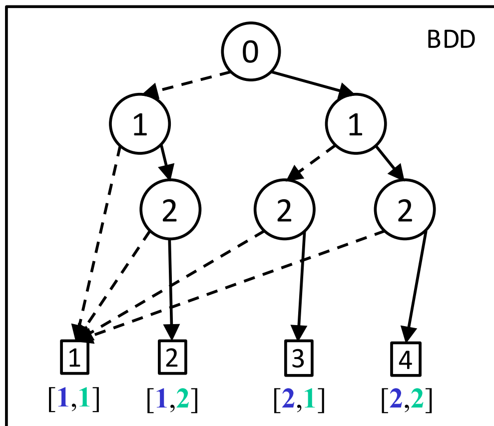
[6]CFLOBDDs github repo: `https://github.com/trishullab/cflobdd`
[7]GCFLOBDDs github repo: `https://github.com/trishullab/GeneralizedCFLOBDDs`



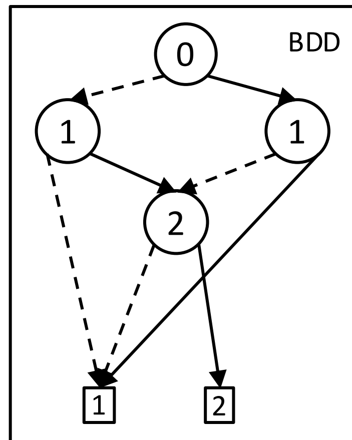
(a) BDD representing a function mapping strings in $\{0,1\}^3$ to indices $[1,2]$. Mapping: $[1 \mapsto \{000, 001, 010, 011, 100, 110\}, 2 \mapsto \{101, 111\}]$.



(b) BDD representing a function mapping strings in $\{0,1\}^3$ to indices $[1,2]$. Mapping: $[1 \mapsto \{000, 001, 100, 101, 010, 110\}, 2 \mapsto \{011, 111\}]$.



(c) PairProduct of Figs. 7.6a and 7.6b with 4 terminal nodes of tuples $[1,1], [1,2], [2,1], [2,2]$.



(d) Effect of Reduce on the BDD in Fig. 7.6c when reduced by the mapping $[[1,1] \mapsto 1, [1,2] \mapsto 2, [2,1] \mapsto 2, [2,2] \mapsto 1]$.

Figure 7.6: Illustration of how PairProduct and Reduce operate on two BDDs that are a part of larger GCFLOBDDs.

both the time taken and the space used (in terms of the number of groupings and edges).

<table>
<tr><th rowspan="2">Benchmark</th><th rowspan="2">#Vars.</th><th colspan="4">CFLOBDD</th><th colspan="5">GCFLOBDD</th></tr>
<tr><th>#Nodes</th><th>#Edges</th><th>Size</th><th>Time(s)</th><th>#Nodes</th><th>#Edges</th><th>Size</th><th>Time(s)</th><th>Grammar</th></tr>
<tr><td rowspan="3">c17</td><td rowspan="3">5</td><td rowspan="3">13</td><td rowspan="3">54</td><td rowspan="3">67</td><td rowspan="3">0.001</td><td>13</td><td>54</td><td>67</td><td>0.001</td><td>Balanced</td></tr>
<tr><td>4</td><td>40</td><td>44</td><td>0.001</td><td>$S_1 \to S_0S_0S_0S_0$
$S_0 \to a$</td></tr>
<tr><td>15</td><td>45</td><td>60</td><td>0.001</td><td>$S_4 \to S_0S_3$
$S_3 \to S_0S_2$
$S_2 \to S_0S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
<tr><td rowspan="3">c432</td><td rowspan="3">36</td><td rowspan="3">2503</td><td rowspan="3">44328</td><td rowspan="3">46831</td><td rowspan="3">0.62</td><td>2503</td><td>44328</td><td>46831</td><td>0.15</td><td>Balanced</td></tr>
<tr><td>683</td><td>21283</td><td>21966</td><td>0.08</td><td>$S_3 \to S_2S_2S_2S_2S_2S_2S_2S_2S_2$
$S_2 \to S_1S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
<tr><td>9</td><td>37483</td><td>37492</td><td>0.18</td><td>$S_1 \to S_0 \ldots S_0$(36 times)
$S_0 \to a$</td></tr>
<tr><td rowspan="3">c880</td><td rowspan="3">60</td><td rowspan="3">1481</td><td rowspan="3">13679</td><td rowspan="3">15160</td><td rowspan="3">0.07</td><td>1481</td><td>13679</td><td>15160</td><td>0.04</td><td>Balanced</td></tr>
<tr><td>988</td><td>11855</td><td>12843</td><td>0.03</td><td>$S_7 \to A_6S_3S_4$
$A_6 \to A_5S_2$
$A_5 \to S_4S_4$
$S_4 \to S_3S_3$
$S_3 \to S_2S_2$
$S_2 \to S_1S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
<tr><td>976</td><td>11821</td><td>12797</td><td>0.03</td><td>$S_6 \to A_5S_2S_3S_4$
$A_5 \to S_4S_4$
$S_4 \to S_3S_3$
$S_3 \to S_2S_2$
$S_2 \to S_1S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
<tr><td rowspan="3">c6288_8</td><td rowspan="3">16</td><td rowspan="3">1974</td><td rowspan="3">22966</td><td rowspan="3">24940</td><td rowspan="3">1.07</td><td>1974</td><td>22966</td><td>24940</td><td>0.36</td><td>Balanced</td></tr>
<tr><td>3259</td><td>20322</td><td>23581</td><td>0.41</td><td>$S_6 \to S_2S_5$
$S_5 \to S_3S_4$
$S_4 \to S_2S_3$
$S_3 \to S_1S_2$
$S_2 \to S_0S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
<tr><td>2008</td><td>23066</td><td>25074</td><td>0.37</td><td>$S_7 \to S_6S_1$
$S_6 \to S_5S_1$
$S_5 \to S_4S_4$
$S_4 \to S_3S_3$
$S_3 \to S_2S_2$
$S_2 \to S_1S_1$
$S_1 \to S_0S_0$
$S_0 \to a$</td></tr>
</table>

Table 7.2: Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits. Note that the rows where the grammar is labeled “Balanced” show the performance of the GCFLOBDD implementation when instantiated with a grammar that makes them equivalent to CFLOBDDs.

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		#Nodes	#Edges	Size	Time(s)	#Nodes	#Edges	Size	Time(s)	Grammar
c6288_9	18	6319	118376	124695	3.86	6319	118376	124695	**1.2**	Balanced
						9991	69707	79698	1.47	$S_6 \rightarrow S_1S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						7489	54122	**61611**	1.32	$S_6 \rightarrow S_3S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
c6288_10	20	6998	409314	416312	7.72	6998	409314	416312	**2.56**	Balanced
						18921	182300	201221	4.49	$S_6 \rightarrow S_1S_3S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						21271	157901	**179172**	4.5	$S_6 \rightarrow S_2S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
c6288_11	22	59110	1107573	1166683	25.23	59110	1107573	1166683	19.46	Balanced
						45188	381797	426985	16.59	$S_7 \rightarrow S_6S_0$ $S_6 \rightarrow S_4S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						45821	424879	**470700**	**15.84**	$S_6 \rightarrow S_3S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$

Table 7.3: Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits (cont.).

7.4.1 RQ1: Can GCFLOBDDs perform better than CFLOBDDs in the case of representing combinatorial circuits?

In §3.9.2.2, we found that BDDs performed better than CFLOBDDs for representing combinatorial circuits because either (1) the benchmarks did not contain any

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		#Nodes	#Edges	Size	Time(s)	#Nodes	#Edges	Size	Time(s)	Grammar
c6288_12	24	326924	3838786	4165710	357.02	326924	3838786	4165710	246	Balanced
						142386	1901957	2044343	86.53	$S_6 \rightarrow S_5S_1$ $S_5 \rightarrow S_4A_4$ $A_4 \rightarrow S_3S_3$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						150262	1290542	**1440804**	**81.15**	$S_7 \rightarrow S_6S_1$ $S_6 \rightarrow S_3S_5$ $S_5 \rightarrow S_3S_4$ $S_4 \rightarrow S_2S_3$ $S_3 \rightarrow S_1S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$

Table 7.4: Table of the performance of CFLOBDDs against GCFLOBDDs with different input grammars on various combinatorial circuits (cont.).

repeated structure that can be exploited by CFLOBDDs, or (ii) the repeated structure in the benchmarks did not fit well with the balanced grammar of CFLOBDDs.

In this section, we investigate whether GCFLOBDDs with a different grammar can represent the circuits better than CFLOBDDs that implicitly assume a balanced grammar. Tabs. 7.2, 7.3, and 7.4 show the performance (in terms of space and time) of GCFLOBDDs on circuits obtained from `iscas85` benchmark suite with different grammars. Note that the rows where the grammar is labeled "Balanced" show the performance of the GCFLOBDD implementation when instantiated with a grammar that makes them equivalent to CFLOBDDs. The performance of Balanced-GCFLOBDDs is slightly better than CFLOBDDs because of a few lower-level implementation level optimizations. Because of the slight performance difference between CFLOBDDs and Balanced-GCFLOBDDs (GCFLOBDDs with "balanced" grammar), in this section, we primarily compare the performance of Custom-GCFLOBDDs (GCFLOBDDs with "custom" grammar) and Balanced-GCFLOBDDs.

We found that GCFLOBDDs with a different grammar structure represent some functions better than CFLOBDDs. For example, by manually inspecting the

Benchmark	#Vars.	GCFLOBDD				BDD	
		#Vertices	#Edges	Size	Time (sec)	Size	Time (sec)
c17	5	4	40	44	**0.001**	**22**	0.004
c432	36	683	21283	**21966**	0.08	27916	**0.01**
c880	60	976	11821	**12797**	0.03	15478	**0.01**
c6288_8	16	3259	20322	23581	1.07	**18516**	**0.02**
c6288_9	18	7489	54122	61611	1.32	**53416**	**0.06**
c6288_10	20	21271	157901	179172	4.5	**150898**	**0.25**
c6288_11	22	45188	381797	426985	16.59	**428190**	**0.97**
c6288_12	24	150262	1290542	1440804	81.15	**1219808**	**3.64**

Table 7.5: Table of the performance of Custom-GCFLOBDDs (the grammar for each benchmark is the best grammar that efficiently represented the circuit in terms of the size among the ones shown in Tabs. 7.2–7.4) against BDDs on different combinatorial circuits.

circuit for the benchmark ‘c432’, we realized that there is a repeated structure to the benchmark that does not fit well into the half-and-half balanced structure of CFL-OBDDs. By using a different grammar, we found that GCFLOBDDs represented c432 better than CFLOBDDs in terms of space and time. We found this trend of GCFLOBDDs with a different grammar representing functions better than Balanced-GCFLOBDDs (and CFLOBDDs) in the case of other benchmarks as well. In particular, in the multiplier circuits like c6288_9, a GCFLOBDD with a Fibonacci grammar structure represented the circuit better than CFLOBDDs by $\sim 3\times$. This shows that Custom-GCFLOBDDs are better than Balanced-GCFLOBDDs (and CFLOBDDs) in the domain of representing combinatorial circuits.

In §3.9.2.3, we discussed how BDDs significantly outperform CFLOBDDs when representing combinatorial circuits. Tab. 7.5 shows the performance of GCFLOBDDs with a grammar that gives the best performance and BDDs in representing combinatorial circuits. We observe that, BDDs still outperform the best [8] GCFLOBDDs on these benchmarks by 13.3% to 50% on most benchmarks reducing from 25.76% to 70.76% that was observed when using CFLOBDDs, in terms of the

[8]the best refers to the custom grammar we identified that works well for the benchmark, there could be a different grammar that is better that has not been identified by us.

size. Interestingly, Custom-GCFLOBDDs represent functions better than BDDs on 'c432', 'c880' and 'c6288_11' benchmarks.

These results show that, although GCFLOBDDs could not outperform BDDs in this domain, Custom-GCFLOBDDs clearly represent functions better than Balanced-GCFLOBDDs (and CFLOBDDs) when instantiated with a different grammar structure that aligns better with the underlying function.

7.4.2 Can GCFLOBDDs represent functions significantly better than CFLOBDDs in terms of space and time?

In this section, we evaluate GCFLOBDDs with different grammars against Balanced-GCFLOBDDs and CFLOBDDs on 10 synthetic benchmarks. These benchmarks are functions that are designed to show that using a grammar other than a "Balanced" grammar significantly affects the representation size and the time taken for constructing the functions. The functions are constructed using mostly Boolean operations, but sometimes arithmetic operations. Some complex functions involve recursive modules for constructing the functions. We will discuss each of the functions in detail:

1. $B_1 : \{0,1\}^n \times \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}$

   $B_1(x_0, y_0, z_0, \ldots, x_{n-1}, y_{n-1}, z_{n-1}) := \bigwedge_{i=0}^{n-1} (x_i \wedge y_i) \vee (\bar{x}_i \wedge z_i)$

2. $B_2 : \{0,1\}^n \rightarrow \{0,1\}$

   $B_2(x_0, \ldots, x_{n-1}) := ((x_0 \oplus x_1) \wedge (\bigwedge_{i=2}^{n-1} x_i)) \vee ((x_0 \bar{\oplus} x_1) \wedge (\bigvee_{i=2}^{n-1} x_i))$

3. Before discussing B_3, let us define few arithmetic operations equivalent to Boolean NOT, AND, XOR.

   $$\begin{aligned} AND_Z(x,y) &= xy \\ NOT_Z(x) &= 1 - x \\ XOR_Z(x,y) &= x + y - 2xy \end{aligned}$$

   $B_3 : \{0,1\}^n \rightarrow \{0,1\}$, where n is of the form $n = 2^k + 4$, $k \geq 1$.

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		Time (s)	Groupings	Edges	Size	Time (s)	Nodes	Edges	Size	Grammar
B_1	177147	15.57	123	410	533	**6.19**	23	94	**117**	$S_{11} \rightarrow S_{10}S_{10}S_{10}$ $S_{10} \rightarrow S_9S_9S_9$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						15.23	123	410	533	Balanced
	531441	93.39	141	474	615	**24.67**	25	102	**127**	$S_{12} \rightarrow S_{11}S_{11}S_{11}$ $S_{11} \rightarrow S_{10}S_{10}S_{10}$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						92.18	141	474	615	Balanced
	1594323	582.08	147	495	642	**126.71**	27	110	**137**	$S_{13} \rightarrow S_{12}S_{12}S_{12}$ $S_{12} \rightarrow S_{11}S_{11}S_{11}$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						578.86	147	495	642	Balanced
B_2	131074	15.06	101	327	428	**10.37**	54	148	**202**	$S_{18} \rightarrow S_1S_{17}$ $S_{17} \rightarrow S_{16}S_{16}$ $S_{16} \rightarrow S_{15}S_{15}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						14.74	101	327	428	Balanced
	262146	42.21	107	346	453	**28.27**	57	156	**213**	$S_{19} \rightarrow S_1S_{18}$ $S_{18} \rightarrow S_{17}S_{17}$ $S_{17} \rightarrow S_{16}S_{16}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						41.55	107	346	453	Balanced
	524290	134.38	113	365	478	**86**	60	164	**224**	$S_{20} \rightarrow S_1S_{19}$ $S_{19} \rightarrow S_{18}S_{18}$ $S_{18} \rightarrow S_{17}S_{17}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						131.94	113	365	478	Balanced

Table 7.6: Performance of GCFLOBDDs with two grammar options – a custom grammar and a balanced grammar – and CFLOBDDs, on synthetic benchmarks with different number of variables. The Balanced grammar option simulates CFLOBDD using GCFLOBDDs.

$$B_3(x_0, \ldots, x_{n-1}) \ := \ AND_Z(AND_Z(F_1, J_1), AND_Z(F_2, J_2))$$

where,

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		Time (s)	Nodes	Edges	Size	Time (s)	Nodes	Edges	Size	Grammar
B_3	524292	46.95	92	329	421	**26.94**	58	163	**221**	$S_{19} \rightarrow S_1S_{18}S_1S_{18}$ $S_{18} \rightarrow S_{17}S_{17}$ $S_{17} \rightarrow S_{16}S_{16}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						46.58	92	329	421	Balanced
	1048580	146.48	97	311	408	**76.75**	61	171	**232**	$S_{20} \rightarrow S_1S_{19}S_1S_{19}$ $S_{19} \rightarrow S_{18}S_{18}$ $S_{18} \rightarrow S_{17}S_{17}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						146.13	97	347	444	Balanced
	2097156	534.23	102	365	467	**245.78**	64	179	**243**	$S_{21} \rightarrow S_1S_{20}S_1S_{20}$ $S_{20} \rightarrow S_{19}S_{19}$ $S_{19} \rightarrow S_{18}S_{18}$ $\vdots$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						529.52	102	365	467	Balanced
B_4	559872	66.93	5561	52942	58503	**29.3**	30	108	**138**	$S_{15} \rightarrow S_{14}S_{14}$ $S_{14} \rightarrow S_{13}S_{13}S_{13}$ $\vdots$ $(alternating)$ $\vdots$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						65.68	5561	52942	58503	Balanced
	1679616	364.11	9260	96407	105667	**136.39**	33	119	**152**	$S_{16} \rightarrow S_{15}S_{15}S_{15}$ $S_{15} \rightarrow S_{14}S_{14}$ $\vdots$ $(alternating)$ $\vdots$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						340.78	9260	96407	105667	Balanced
	3359232	1390.03	12891	141017	153908	**579.02**	34	122	**156**	$S_{17} \rightarrow S_{16}S_{16}$ $S_{16} \rightarrow S_{15}S_{15}S_{15}$ $\vdots$ $(alternating)$ $\vdots$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						1445.43	12891	141017	153908	Balanced

Table 7.7: Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.).

$$
\begin{aligned}
F_1 &:= AND_Z(x_0, x_1) - NOT_Z(AND_Z(x_0, x_1)) \\
J_1 &:= \Pi_{t=1}^{\frac{n-4}{4}} XOR_Z(x_t, x_{t+1}) \\
F_2 &:= AND_Z(x_{n/2}, x_{n/2+1}) - NOT_Z(AND_Z(x_{n/2}, x_{n/2+1})) \\
J_2 &:= \Pi_{t=1}^{\frac{n-4}{4}} XOR_Z(x_{n/2+t}, x_{n/2+t+1})
\end{aligned}
$$

4. We will define function B_4 recursively.

$$
\begin{aligned}
\Phi(x, y) &= x \oplus y \\
\Psi(x, y, z) &= (x \wedge y) \vee (\bar{x} \wedge z)
\end{aligned}
$$

   $B_4 : \{0,1\}^n \to \{0,1\}$, such that $n = 3^p 2^q, p > 0, q > 0$. Let us define a family of functions g_i as follows:

$$
x_0, \ldots, x_{n-1} \in \{0,1\}, \qquad g_j^{(0)} = x_j.
$$

$$
g_m^{(t+1)} = \begin{cases} \Phi\big(g_{2m}^{(t)},\ g_{2m+1}^{(t)}\big), & t \text{ is even}, 0 \le m < \frac{|g^{(t}|}{2} \\ \Psi\big(g_{3m}^{(t)},\ g_{3m+1}^{(t)},\ g_{3m+2}^{(t)}\big), & t \text{ is odd}, 0 \le m < \frac{|g^{(t}|}{3} \end{cases}
$$

   and, $B_4(x_0, \ldots, x_{n-1}) := g_0^{p+q}$.

5. $B_5 : \{0,1\}^n \to \{0,1\}$ such that $n = fib[k], k > 1$, where fib is the Fibonacci sequence: $fib = [0, 1, 1, 2, 3, \ldots, n]$.

   Let us define another function $G(n, s)$ as;

$$
G(n, s) = \begin{cases} \neg x_0, & fib[n] = 0, \\ x_s, & fib[n] = 1, \\ x_s \oplus x_{s+1}, & fib[n] = 2, \\ x_s \wedge (x_{s+1} \oplus x_{s+2}), & fib[n] = 3, \\ G(n-2, s) \ \wedge \ G\big(n-1,\ s + fib[n-2]\big), & fib[n] \ge 4. \end{cases}
$$

   .

   We will define B_5 using $G(n, s)$.

   $B_5(x_0, \ldots, x_{n-1}) := (x_0 \wedge (x_1 \oplus x_2)) \wedge \bigwedge_{i=5}^{|fib|-2} G(i, fib[i-1])$

6. $B_6 : \{0,1\}^{k*n} \to \{0,1\}$. We define B_6 using a group of sub-functions g_i as follows:

$$x_0, \ldots, x_{kn-1} \in \{0,1\}, \qquad g_i = \bigoplus_{j=0}^{n-1} x_{in+j} \quad (0 \le i < k).$$

$$\mathrm{Op}_t = \begin{cases} \text{AND} & t \equiv 0 \pmod 5, \\ \text{OR} & t \equiv 1 \pmod 5, \\ \text{XOR} & t \equiv 2 \pmod 5, \\ \text{NOR} & t \equiv 3 \pmod 5, \\ \text{NAND} & t \equiv 4 \pmod 5, \end{cases}$$

$$F = \bigwedge_{\substack{i=0 \\ i \text{ even}}}^{k-2} \big(\mathrm{Op}_i(g_i, g_{i+1}) \;\vee\; \mathrm{Op}_{i+1}(g_i, g_{i+1})\big)\,.$$

7. $B_7 : \{0,1\}^n \to \{0,1\}$

$$\begin{aligned} B_7(x_0, \ldots, x_{n-1}) &:= (x_0 \wedge F_1) \vee (\bar{x_0} \wedge F_2) \\ F_1 &:= \bigwedge_{t=0}^{\lfloor \frac{n-2}{3} \rfloor} ((x_{3t+1} \wedge x_{3t+2} \wedge \bar{x}_{3t+3}) \vee (\bar{x}_{3t+1} \wedge \bar{x}_{3t+2} \wedge x_{3t+3}) \\ F_2 &:= \bigwedge_{t=0}^{\lfloor \frac{n-2}{3} \rfloor} ((x_{3t+1} \wedge (x_{3t+2} \oplus x_{3t+3})) \vee (\bar{x}_{3t+1} \wedge (\overline{x_{3t+2} \oplus x_{3t+3}}))) \end{aligned}$$

8. $B_8 : \{0,1\}^n \to \{0,1\}$ Let us consider a group schedule $\mathbf{g}$, an array of length T such that $\Pi_{i=0}^{T-1} g_i = n$.

$$\mathbf{g} = (g_0, \ldots, g_{T-1}), \qquad x_0, \ldots, x_{N-1} \in \{0,1\}.$$

In this function, we consider a sequence of prime numbers as $\mathbf{g}$, i.e., $\mathbf{g} = \{2, 3, 5, 7, \ldots\}$. We will now define a series of sub-functions on which B_8 is defined recursively.

- We define a cyclic-choice operator Op as

$$\mathrm{Op}(i) = \begin{cases} \wedge, & i \equiv 0 \pmod 5, \\ \vee, & i \equiv 1 \pmod 5, \\ \oplus, & i \equiv 2 \pmod 5, \\ \mathrm{NOR}, & i \equiv 3 \pmod 5, \\ \mathrm{NAND}, & i \equiv 4 \pmod 5, \end{cases}$$

  and alternating negation $y^{(s,p)}$ as

$$y_i^{(s,p)} = \begin{cases} \neg x_{s+i}, & i \text{ even}, \\ x_{s+i}, & i \text{ odd}, \end{cases} \qquad i = 0, \ldots, p-1.$$

- We define a function $\mathrm{S}(s,p)$ as:

$$\begin{aligned} A_0^{(s,p)} &= y_0^{(s,p)} \\ A_i^{(s,p)} &= \mathrm{Op}(i)\Big(A_{i-1}^{(s,p)},\, y_i^{(s,p)}\Big) \quad (i = 1, \ldots, p-1) \\ \mathrm{S}(s,p) &= A_{p-1}^{(s,p)} \end{aligned}$$

- We define another series of functions $F_m^{(t)}$ as follows. Every series $F_m^{(t)}$ is a sequence of m elements in "round" t. $F_m^{(0)}$ is defined over variables $x_0, \ldots, x_{n-1}$ and $m = n$ but for $t > 0$, $F_m^{(t+1)}$ is evaluated over elements in the previous sequence – $F_m^{(t)}$.

$$\begin{aligned} F_j^{(0)} &= x_j & \forall j = 0...n-1 \\ F_m^{(t+1)} &= \mathrm{S}\Big(s = 0, p = g_t\Big)\Big|_{x_i \mapsto F^{(t)}_{mg_t+i}} \end{aligned}$$

  This implies that for $F_m^{(t+1)}$, S operates elements in sequence $F_m^{(t)}$. Finally,

$$B_8 = F_0^{(T)}.$$

9. To define, $B_9 : \{0,1\}^n \to \{0,1\}$, we use the same approach as B_8, except that the group sequence $\mathbf{g}$ is a sequence of odd numbers.

$$\mathbf{g} = \{3, 5, 7, 9, \ldots, \}$$

10. Similarly, $B_{10} : \{0,1\}^n \to \{0,1\}$ is defined using the same approach as B_8 with a custom group sequence $\mathbf{g}$.

$$\mathbf{g} = \{5, 2, 6, 4, 3, 15, 3, 6\}$$

Note that the sequence is considered until the product of the elements in the sequence is equal to the number of variables in the function.

Tabs. 7.6–7.10 show the performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks B_1 to B_{10} with increasing number of variables. GCFLOBDDs are instantiated with two different grammar types – (1) a custom grammar that we identified, which efficiently represents the benchmark, and (2) a balanced grammar, which effectively simulates CFLOBDDs (and controls for the differences in the two different codebases). We found that Custom-GCFLOBDDs, i.e., GCFLOBDDs with a custom grammar, represent their corresponding benchmarks much more efficiently than Balanced-GCFLOBDDs in terms of size, with an improvement ranging from 47.51% to 99.9%. In terms of the time taken for constructing the functions, Custom-GGCFLOBDDs outperform Balanced-GCFLOBDDs on most benchmarks with a speedup ranging from 0.78× to 4.59×. In particular, CFLOBDDs perform modestly better than Custom-GCFLOBDDs on the B_5 benchmark. In all other cases, Custom-GCFLOBDDs perform better than Balanced-GCFLOBDDs (and CFLOBDDs) in terms of space and time.

These experiments show that GCFLOBDDs are useful in capturing repeatedness and structural similarities that go beyond the "balanced" hierarchy seen in CFLOBDDs, and as a result can hopefully be useful in a broader set of domains.

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		Time (s)	Nodes	Edges	Size	Time (s)	Nodes	Edges	Size	Grammar
B_5	196418	12.64	1762	5947	7709	12.49	328	828	**1156**	$S_{25} \rightarrow S_{23}S_{24}$ $S_{24} \rightarrow S_{22}S_{23}$ $\vdots$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						12.43	1762	5947	7709	Balanced
	317811	**26.23**	2239	7548	9787	27.07	354	895	**1249**	$S_{26} \rightarrow S_{24}S_{25}$ $S_{25} \rightarrow S_{23}S_{24}$ $\vdots$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						27.22	2239	7548	9787	Balanced
	514229	**43.12**	2838	9553	12391	55.63	381	961	**1342**	$S_{27} \rightarrow S_{25}S_{26}$ $S_{26} \rightarrow S_{24}S_{25}$ $\vdots$ $S_2 \rightarrow S_0S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						45.51	2838	9553	12391	Balanced
B_6	40000	2.58	510	1870	2380	**1.06**	29	250	**279**	$S_8 \rightarrow S_7S_7S_7S_7$ $S_7 \rightarrow S_6S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						2.56	510	1870	2380	Balanced
	90000	8.02	1047	3769	4816	**2.7**	44	411	**455**	$S_9 \rightarrow S_8S_8S_8$ $S_8 \rightarrow S_7S_7S_7$ $S_7 \rightarrow S_6S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						7.9	1047	3769	4816	Balanced
	250000	35.89	1692	6102	7794	**8.24**	20	120	**140**	$S_9 \rightarrow S_8S_8S_8S_8S_8$ $S_8 \rightarrow S_7S_7S_7S_7S_7$ $S_7 \rightarrow S_6S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						33.85	1692	6102	7794	Balanced

Table 7.8: Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.)

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		Time (s)	Nodes	Edges	Size	Time (s)	Nodes	Edges	Size	Grammar
B_7	59050	9.38	215	786	1001	**5.78**	32	141	**173**	$S_{11} \rightarrow S_0S_{10}$ $S_{10} \rightarrow S_9S_9S_9$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						8.95	215	786	1001	Balanced
	177148	53.78	229	830	1059	**25.41**	35	154	**189**	$S_{12} \rightarrow S_0S_{11}$ $S_{11} \rightarrow S_{10}S_{10}S_{10}$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						50.9	229	830	1059	Balanced
	531442	451.66	272	977	1249	**147.54**	38	167	**205**	$S_{13} \rightarrow S_0S_{12}$ $S_{12} \rightarrow S_{11}S_{11}S_{11}$ $\vdots$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						415.5	229	830	1059	Balanced
B_8	2310	**0.09**	481	2023	2504	**0.09**	11	76	**87**	$S_5 \rightarrow S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						0.09	481	2023	2504	Balanced
	30030	1.26	2323	11207	13530	1.45	13	112	**125**	$S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						1.14	2323	11207	13530	Balanced
	510510	39.77	13018	73116	86134	**37.65**	15	158	**173**	$S_7 \rightarrow S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1$ $S_1 \rightarrow S_0S_0$ $S_0 \rightarrow a$
						39.56	13018	73116	86134	Balanced

Table 7.9: Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.)

Benchmark	#Vars.	CFLOBDD				GCFLOBDD				
		Time (s)	Nodes	Edges	Size	Time (s)	Nodes	Edges	Size	Grammar
B_9	945	0.04	411	1779	2190	**0.03**	9	67	**108**	$S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						0.04	411	1779	2190	Balanced
	10395	0.55	1915	9080	10995	**0.42**	11	97	**108**	$S_5 \rightarrow S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						0.49	1915	9080	10995	Balanced
	135135	10.69	8512	47200	55712	**7.08**	13	133	**146**	$S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3S_3S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1S_1S_1S_1$ $S_1 \rightarrow S_0S_0S_0$ $S_0 \rightarrow a$
						10.13	8512	47200	55712	Balanced
B_{10}	10800	0.54	748	4196	4944	0.52	13	92	**105**	$S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0S_0S_0S_0$ $S_0 \rightarrow a$
						0.48	748	4196	4944	Balanced
	32400	1.62	1154	7207	8361	**1.1**	15	112	**127**	$S_7 \rightarrow S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0S_0S_0S_0$ $S_0 \rightarrow a$
						1.42	1154	7207	8361	Balanced
	194400	14.33	3243	24185	27428	**7.34**	17	126	**146**	$S_8 \rightarrow S_7S_7S_7S_7S_7S_7$ $S_7 \rightarrow S_6S_6S_6$ $S_6 \rightarrow S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5S_5$ $S_5 \rightarrow S_4S_4S_4$ $S_4 \rightarrow S_3S_3S_3S_3$ $S_3 \rightarrow S_2S_2S_2S_2S_2S_2$ $S_2 \rightarrow S_1S_1$ $S_1 \rightarrow S_0S_0S_0S_0S_0$ $S_0 \rightarrow a$
						13.94	3243	24185	27428	Balanced

Table 7.10: Performance of GCFLOBDDs and CFLOBDDs on synthetic benchmarks (cont.)

Chapter 8: Related Work

8.1 Comparison with BDD variants

Over the years, many variants of BDDs have been proposed Sasao and Fujita (1996). These data structures can be broadly divided into three families: ones that make use of weights on edges, ones that do not use edge weights, and ones that allow the underlying graph to have cycles.

Unweighted BDD variants include Multi-Terminal BDDs Clarke et al. (1993, 1995), Algebraic Decision Diagrams Bahar et al. (1997), Free Binary Decision Diagrams (FBDDs) (Wegener, 2000, §6), Binary Moment Diagrams (BMDs) Bryant and Chen (1995), Hybrid Decision Diagrams (HDDs) Clarke et al. (1995), Differential BDDs Anuchitanukul et al. (1995), and Indexed BDDs (IBDDs) Jain et al. (1997). Several of these BDD variants offers exponential compression over classical BDDs. However, because FBDDs, BMDs, HDDs, and IBDDs that encode, e.g., the identity function, need to examine each variable, the exponential-separation argument for CFLOBDDs from §3.7 carries over for all of these variants.

Cyclic BDD variants include Linear/Exponentially Inductive Functions (LIFs/EIFs) Gupta and Fisher (1993); Gupta (1994) and Cyclic BDDs (CBDDs) Reffel (1999). Because they allow cycles, CFLOBDDs are closer to the structures in this category. The differences between CFLOBDDs and these representations can be characterized as follows:

- The aforementioned representations all make use of numeric/arithmetic annotations on the edges of the graphs used to represent functions over Boolean arguments, rather than the matched-path principle that is the basis of CFLOBDDs. Matched paths can be characterized in terms of a context-free language of matched parentheses, rather than in terms of numbers and arithmetic (see Eqn. (3.1)).

- An essential part of the design of LIFs and EIFs is that the BDD-like subgraphs in them are connected in very restricted ways. In contrast, in CFLOBDDs, different groupings at the same level (or different levels) can have very different kinds of connections in them.
- Similarly, CBDDs require that there be some fixed BDD pattern that is repeated over and over in the structure; a given function uses only a few such patterns. With CFLOBDDs, there can be many reused patterns (i.e., in the lower-level groupings in CFLOBDDs).
- CBDDs are not canonical representations of Boolean functions, which complicates the algorithms for performing certain operations on them, such as the operation to determine whether two CBDDs represent the same function.
- The layering in CFLOBDDs serves a different purpose than the layering found in LIFs/EIFs and CBDDs. In the latter representations, a connection from one layer to another serves as a jump from one BDD-like fragment to another BDD-like fragment. In CFLOBDDs, only the lowest layer (i.e., the collection of level-0 groupings) consists of BDD-like fragments (and just two very simple ones at that); it is only at level 0 that the values of variables are interpreted. As one follows a matched path through a CFLOBDD, the connections between the groupings at levels above level 0 serve to encode which variable is to be interpreted next.

LIFs/EIFs/CBDDs could be generalized by replacing BDD-like subgraphs in them with CFLOBDDs.

Examples of (acyclic) edge-weighted BDD variants include Edge-Valued BDDs (EVBDDs) Vrudhula et al. (1996), Factored Edge-Valued BDDs (FEVBDDs) Tafertshofer and Pedram (1997), Quantum Information Decision Diagram (QuIDDs) Viamontes et al. (2003), Quantum Multiple-valued DDs (QMDDs) Niemann et al. (2016), MQTDDs Zulehner et al. (2019), and Tensor Decision Diagrams (TDDs)

Hong et al. (2020), in which the edges out of decision nodes have weights, and some "accumulation" (e.g., $+$ or $\times$) of the weights encountered on the path for an assignment a yields the function's value on a. QMDDs, QuIDDs, MQTDDs, and TDDs are all variants what we have been calling WBDDs (with complex-valued weights). They are based on a Shannon decomposition, and thus the only sharing is of sub-DAGs, whereas the call-return structure used in WCFLOBDDs allows sharing of the "middle of a DAG." If the weights are allowed to be unboundedly large, a polynomial-sized data structure of this sort can be used to encode a decision tree that is double-exponentially larger. However, to the best of our knowledge, such double-exponential compression is impossible when the weights are required to use a constant number of bits.

Other data structures that generalize BDDs are representations like Sentential Decision Diagrams (SDDs) Darwiche (2011) and Variable Shift SDDs Nakamura et al. (2020). These data structures generalize BDDs by assuming a tree-shaped ordering over variables, and there are functions for which these data structures offer double-exponential compression over decision trees and an exponential compression over BDDs. SDDs and VS-SDDs represent Boolean functions only ($\mathbb{B}^n \rightarrow \mathbb{B}$), but have also been extended to represent probability distributions Kisa et al. (2014). In CFLOBDDs (and WCFLOBDDs, GCFLOBDDs), a grouping g can have multiple middle vertices that reuse the same B-connection grouping b, as long as the return edges for the different invocations of b use different mappings to g's exit vertices. This "contextual rewiring" gives CFLOBDDs (and its variants) greater ability to reuse substructures than SDDs and VS-SDDs. (Moreover, b can also be used as the A-connection grouping of g.) VS-SDDs support a sub-structure sharing property that goes beyond SDDs; however, as with BDDs and WBDDs, the objects that are shared are always sub-DAGs. In contrast, the (W)CFLOBDD "procedure-call" device is a more flexible mechanism for reusing substructures than the mechanism for reusing substructures in VS-SDDs. In particular, the call-return structure used in (W)CFLOBDDs allows sharing of the "middle of a DAG." SDDs and VS-SDDs (and

their quantitative generalizations, such as Probabilistic SDDs Kisa et al. (2014)) have not, so far, been used in matrix computations, and implementations of operations such as Kronecker product and matrix multiplication based on these structures are unknown, which meant that we could not use them in our quantum-simulation experiments. We did compare CFLOBDDs against SDDs for two of the micro-benchmarks, and found that CFLOBDDs were much faster (Tab. 3.2). However, the relationship between these representations and CFLOBDDs merits future study.

Another approach to introducing hierarchy into BDDs is the notion of *Tree BDDs* McMillan (1994). Tree BDDs employ a hierarchical representation that closely resembles a tree automaton. In this framework, the variable set of a Boolean function is recursively partitioned into three components: root variables, left-child variables, and right-child variables. The left and right partitions recursively represent their corresponding sub-functions using the same construction, while the root partition combines the equivalence classes induced by the left and right sub-functions to define the overall function. A key distinction from CFLOBDDs is that, in Tree BDDs, each partition explicitly associates variables with the corresponding sub-functions. In contrast, CFLOBDDs reuse substructures via the *Contextual Interpretation Principle*, wherein groupings are not explicitly tied to particular variables. Even if the Tree BDD framework were modified to eliminate explicit variable instantiation, the resulting structures would still be closely aligned with tree automata and could therefore encounter challenges similar to those faced by TIDDs, due to the absence of a linear structure.

Local Invertible Map DDs (LIMDDs) Vinkhuijzen et al. (2023a) are a DAG-structured decision diagram (i.e., no "procedure calls") in which two nodes are merged if their respective decision trees (considered as vectors) are equal up to multiplication by a tensor product of 2×2 matrices that represent single-qubit gates. It has been shown that they can be exponentially more succinct that WBDDs—results that should carry over to WCFLOBDDs (i.e., there are functions for which LIMDDs are

efficient, but WCFLOBDDs are not). One drawback of LIMDDs is that every node-creation operation requires performing a cubic-time operation akin to Gaussian elimination to maintain the invariants that ensure canonicity; consequently, LIMDDs seem to have a built-in handicap. Each of the steps required to ensure that WCFLOBDDs are canonical are very fast in comparison (e.g., hashed lookups). Little has been published about the performance of LIMDDs, but one empirical evaluation Vinkhuijzen et al. (2023b) reported that QFT on a random stabilizer state took about 250,000 seconds for 24 qubits.

8.2 Relationship to PDSs, NWAs, etc.

As noted in §3.1, CFLOBDDs can be seen as a generalization of BDDs in which a restricted form of procedure call is permitted. As such, CFLOBDDs are similar to various structures used in the model-checking community, namely, Hierarchical FSMs (HFSMs) Alur et al. (2005a), Push Down Systems (PDSs) Bouajjani et al. (1997); Finkel et al. (1997), and Nested-Word Automata (NWA) Alur and Madhusudan (2009), which all have the same flavor of "graph plus procedure calls."

- As discussed in §6.1.1, there is a formal sense in which a CFLOBDD is an NWA.
- HFSMs, PDSs, and NWAs have mainly been investigated for modeling infinite-state systems. What CFLOBDDs demonstrate is that there are advantages to considering finite restrictions of such structures. In particular, the advantage of introducing a procedure-call-like mechanism in a finite model is that one can obtain an exponential-factor advantage compared to the original finite models without procedure calls.

We believe that there is great potential for exploring other combinations of the idea of "BDDs plus procedure calls" that use ideas from HFSMs, PDSs, and NWAs in ways that are different than those found in CFLOBDDs.

CFLOBDDs (and GCFLOBDDs) can be considered to be a special case of Visibly Pushdown Automata (VPAs) Alur and Madhusudan (2004) where A-connections and B-connections in CFLOBDDs, and Connections in general in GCFLOBDDs, correspond to call transitions in a visibly pushdown language (VPL), return edges correspond to return transitions in the VPL, and edges at level-0 correspond to internal transitions of the VPL. There are other classes of VPAs, such as k-module single-entry VPAs (k-SEVPAs) Alur et al. (2005b) and Modular VPAs Kumar et al. (2006), that have minimal canonical representations. Such VPAs have modular components that are similar to the groupings in CFLOBDDs. However these classes of VPAs assume that the modular decomposition is fixed ahead of time. For instance, each k-SEVPA has exactly $k+1$ modules. In contrast, CFLOBDDs use a decomposition that is based on a fixed number of levels, but the specific number of groupings for a function is a by-product of the structural invariants (§3.3.1 and Construction 1 in Appendix §C).

8.3 Prior Approaches to Quantum Simulation

Also related are prior methods for quantum simulation. Such simulation can be exact or approximate; our focus here is on exact simulation (modulo floating-point round-off). Decision diagrams used for such simulation include QMDDs Miller and Thornton (2006); Zulehner and Wille (2019) and TDDs Hong et al. (2020). We also compared our approach to tensor networks, a widely used approach to quantum simulation that is not based on decision diagrams. As shown in Tab. 3.5, CFLOBDDs perform better than tensor networks on some algorithms (GHZ, BV, DJ, Grover) and worse on others (Simon, QFT, Shor).

Similar to the well-known quantum algorithms discussed in this thesis, variational quantum algorithms, which include a noise channel, can also be simulated using CFLOBDDs. Huang et al. Huang et al. (2021) simulate variational quantum algorithms using knowledge-compilation techniques. In their approach, the noise component is modeled as an additional operator whose action is represented as a matrix.

The noise matrix can be represented as a CFLOBDD, and hence CFLOBDDs can also be used for simulating variational quantum algorithms.

8.4 Compression of Programs and Compression Principles

A CFLOBDD can compactly represent many finite paths. This property is akin to a statement that the use of nonrecursive procedures in programs can enable small programs to have many execution paths, and is the essence of the observation in the paper by Melski and Reps Melski and Reps (1999) that an acyclic, non-recursive, interprocedural control-flow graph of size k could have 2^{2^k} matched paths. Although not formulated as a theorem, this observation was stated in Melski's Ph.D. thesis (Melski, 2002, §3.5.4). Melski uses Yannakakis's notion of L-reachability Yannakakis (1990) (i.e., a path from node s to node t only counts as a valid s-t connection if the path's labels form a word in L), and defines the notion of a "finite-path graph" with respect to some language L: there are only a finite number of L-paths from, e.g., program entry to program exit (Melski, 2002, §3.4). He then defines an interprocedural control-flow graph, denoted by G^*_{fin}. One of the languages of interest is the language of unbalanced-left paths (Melski and Reps, 1999, §2.1), in which each return-edge is matched with the closest preceding unmatched call-edge, but there can be zero or more umatched call-edges. (The unbalanced-left language is typically the language of interest for context-sensitive interprocedural dataflow analysis Reps et al. (1995); Reps (1997).) Melski observes, "... the number of [unbalanced-left] paths through

G^*_{fin} can be doubly exponential in the size of G^*_{fin}." [1]

These results are tantamount to the statement (proposed by one of the referees of Sistla et al. (2024)) that "there is a family of programs P_n, written with non-recursive procedures, that each would be exponentially larger if written without non-recursive procedures." In the 1970s, the literature on program schematology Paterson and Hewitt (1970) explored the relative power of various programming constructs, beginning with results showing that recursive procedure calls are more expressive than iteration (in particular, there are recursive program schemes such that, for some interpretation of the function and predicate symbols, any flowchart scheme will produce results that are different from those obtained with the recursive scheme Paterson and Hewitt (1970); Lynch and Blum (1979)). Thus, it would have been natural for the schematology literature to contain a result of the form stated above. However, we were unable to find a paper with such a result; when procedures are allowed, the main interest seems to be in recursive procedures and how such programs compare with programs written in a language without procedure calls, but other features, such as arrays, stacks, or counters Constable and Gries (1972); Garland and Luckham (1973).

The compression abilities of CFLOBDDs are based what might be called "multiplicative amplification": calls to procedures P and Q, when performed in sequence, result in a structure in which the number of $L(\mathit{matched})$-paths is equal to the product of the numbers of $L(\mathit{matched})$-paths through P and Q. Multiplicative amplification

[1] Melski had found that one 20,000-line program had about $2^{400{,}000}$ paths, which prompted Melski and Reps to realize that they were facing double-exponential explosion. Unfortunately, that work was carried out after their CC '99 paper had been published Melski and Reps (1999). The latter paper states, incorrectly, "In the worst case, the number of paths through a program is exponential in the number of branch statements b ..." (Melski and Reps, 1999, §5). (This kind of mistake seems to be common among authors working with structures that are DAG-like, but are really based on acyclic hyper-graphs: they erroneously think that they are dealing with DAGs and conclude that there is exponential explosion/compression, whereas the true state of affairs is that they have *double*-exponential explosion/compression. Examples are found in the literature on E-graphs Nandi et al. (2021); Willsey (2021) and version-space algebras Polozov and Gulwani (2015); Gulwani et al. (2017); Koppel (2021).)

leads to the repeated squaring we see in counting the number of paths from the entry vertex to the (one) exit vertex of a no-distinction proto-CFLOBDD with k levels:

$$P(0) = 1 \qquad\qquad P(n+1) = P(n)^2.$$

A more powerful compression principle—again based on an "amplification" step repeated some number of times—is found in Mairson's rational reconstruction of a proof of Statman's Mairson (1992). As with CFLOBDDs, there are a finite number of stratification levels and no recursion, but instead of "multiplicative amplification," Mairson uses "powerset amplification." He is interested in representing all values of the stratified types defined by

$$\mathcal{D}_0 = \{\text{true}, \text{false}\} \qquad\qquad \mathcal{D}_n = \mathit{powerset}(D_{n-1}).$$

Mairson observes that one can use linked lists to represent the elements of each of the $\mathcal{D}_i$. To represent them concisely, he defines a powerset-combinator `powerset` that takes a list l_1 as input, and returns a list l_2 that contains the powerset of the elements of l_1 (a simple exercise in functional programming). He can then represent $\mathcal{D}_n$ with a λ-calculus term D_n that applies `powerset` n times to the list $\{\text{true}, \text{false}\}$. Considered as a member of the family of terms D_0, $D_1 = \texttt{powerset}(D_0)$, $\ldots$, $D_n = \texttt{powerset}^n(D_0)$, $\ldots$, the size of the term D_n is $\Omega(n)$. In contrast, the size of the set that is represented by D_n is described by the following recurrence relation:

$$S(0) = 1 \qquad\qquad S(n+1) = 2^{S(n)},$$

whose solution is non-elementary: $S(n)$ is an exponential tower of 2s, $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{n}$, of height n.

Chapter 9: Conclusion

This dissertation introduces a new class of decision diagrams aimed at achieving succinct representations of functions. Prior work on Binary Decision Diagrams (BDDs), a widely used representation for efficient manipulation of Boolean and pseudo-Boolean functions showed success across a wide range of domains, including hardware and software verification, program analysis, model checking, network verification, and, more recently, quantum computing. Building on this rich line of work, this thesis demonstrates that further compression can be achieved by explicitly introducing hierarchy into the structure of decision diagrams.

The overall theme of this work is capturing recurring patterns in Boolean and pseudo-Boolean functions by introducing hierarchical structures into decision-diagram representations for succinctness. In Chapter 3, we introduced Context-Free-Language Ordered Binary Decision Diagrams (CFLOBDDs), which add structured hierarchy to decision trees. This hierarchy missing in BDDs helps CFLOBDDs in capturing and reusing common substructures, yielding a more efficient representation in some circumstances. The hierarchy in CFLOBDDs can be considered as sharing intermediate portions of a DAG — reusing "middle-of-a-DAG" structures.

In the best case, we show that CFLOBDDs are exponentially more succinct than BDDs, and double-exponentially more succinct than decision-tree representations of Boolean functions. In particular, a CFLOBDD with k levels can represent Boolean functions over 2^k variables and succinctly encode up to 2^{2^k} paths. As discussed in Chapter 3, a CFLOBDD is composed of groupings, vertices, and edges that satisfy specific structural invariants, ensuring canonicity of the representation. Groupings correspond to subfunctions, while vertices and edges describe how these subfunctions are composed to form the overall function. This hierarchical organization is achieved via a balanced half-and-half decomposition of variables at each level.

Such a decomposition enables CFLOBDDs to capture and reuse repeated patterns across the function, allowing common substructures to be shared.

When we applied CFLOBDDs to the problem of quantum simulation, we observed that CFLOBDDs perform significantly better than BDDs, in terms of compactness of representation and also execution time on most benchmarks. With a 15-minute timeout, the number of qubits that CFLOBDDs can handle are 65,536 for GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover's Algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

Given that CFLOBDDs perform significantly better than BDDs by introducing hierarchy, we explored whether the linear structure in CFLOBDDs also contributed to the succinctness. In Chapter 6, we investigated whether CFLOBDDs fundamentally rely on a linear structure, or whether they are more akin to a restricted class of tree automata that employ hierarchy solely for representational purposes. To understand this issue better, we designed a new data structure, called *Tree-Automata Inspired Decision Diagrams* (TIDDs), and studied the relative expressive power of CFLOBDDs and TIDDs. We observed that although TIDDs, like CFLOBDDs, exhibit exponential separation with respect to BDDs for some of the same families of functions, this behavior does not extend to the general case. We provided intuition and evidence explaining why this separation fails to hold more broadly. This difference arises because the linear structure of CFLOBDDs enables a disciplined decomposition of a function into subfunctions, followed by their systematic composition according to a prescribed linear ordering. Each subfunction can be represented efficiently in isolation, and this ordered composition ensures that these compact representations are reused effectively, resulting in an efficient representation of the overall function. We showed that there exists a family of functions for which, under the same variable ordering, CFLOBDDs are exponentially more succinct than TIDDs.

To understand whether this difference affects performance in practice, we applied CFLOBDDs and TIDDs to the same domain of quantum simulation. We ob-

served that TIDDs performed poorly, primarily due to substantial blow-ups in the sizes of the representations of intermediate state vectors and unitary matrices. These findings indicate that the linear structure in CFLOBDDs indeed helps in efficient function representations.

However, when applied to the representation of combinatorial circuits, we observed that BDDs often outperform CFLOBDDs. One contributing factor is the strict half-and-half decomposition enforced at every level of a CFLOBDD, or, equivalently, the use of a fixed balanced grammar throughout the representation. Such rigid decompositions may fail to align with the inherent structural regularities of a given Boolean function, thereby limiting opportunities for sharing and compression. To address this limitation, we introduced *Generalized Context-Free-Language Ordered Binary Decision Diagrams* (GCFLOBDDs), which extend CFLOBDDs by allowing the use of arbitrary user-specified grammars. In Chapter 7, we formally define the structure of the grammars that can be provided by users, the GCFLOBDD data structure, and the GCFLOBDD operations that are supported.

We experimentally compared GCFLOBDDs with CFLOBDDs and BDDs on combinatorial circuits and synthetic benchmark functions. We observed that, in certain cases, GCFLOBDDs instantiated with suitably chosen custom grammars represent combinatorial circuits more compactly than BDDs. In other cases, BDDs continue to outperform GCFLOBDDs, particularly when the underlying functions exhibit little or no repeated structural pattern. Nevertheless, even in such scenarios, GCFLOBDDs consistently achieved better compression than CFLOBDDs (i.e., GCFLOBDDs with a balanced grammar), demonstrating the benefit of relaxing the fixed balanced-grammar constraint. On synthetic benchmarks, GCFLOBDDs with custom grammars substantially outperformed CFLOBDDs in representation size, achieving reductions ranging from 47.51% to 99.9%, along with execution-time improvements ranging from 0.78× to 4.59×. These results demonstrate that GCFLOBDDs instantiated with custom grammars can represent functions more ef-

ficiently than CFLOBDDs when the repeated structural patterns in those functions do not align with the fixed balanced decomposition enforced by CFLOBDDs.

In Chapter 4, we explored the extension of CFLOBDDs with edge weights to represent efficiently functions with large output ranges. For functions of the form $h : \{0,1\}^n \rightarrow V$, where $V \subseteq D$ is a large set, the size of a CFLOBDD that represents h is necessarily at least $|V|$. Consequently, both BDDs and CFLOBDDs struggle to represent such functions efficiently. In the context of BDDs, this limitation is addressed through *Weighted BDDs* (WBDDs), which associate weights with edges and compute the value of a function on a given assignment by composing the weights along the corresponding path. We extended this idea to CFLOBDDs and studied how adding weights interacts with their hierarchical structure, including how key properties such as canonicity and algorithms work. To this end, we introduced *Weighted Context-Free-Language Ordered Binary Decision Diagrams* (WCFLOBDDs), which combine the hierarchical decomposition of CFLOBDDs with the expressive power of weighted edges. This hybrid representation enables efficient encoding of a broader class of functions than either CFLOBDDs or BDDs alone. We showed that there exists a family of functions for which WCFLOBDDs are exponentially more succinct than both CFLOBDDs and BDDs.

This "best of both worlds" behavior is further illustrated by our empirical evaluation of WCFLOBDDs in the domain of quantum simulation. Under a 15-minute timeout, WCFLOBDDs successfully simulate circuits with up to 1,048,576 qubits for GHZ states (a 1× improvement over CFLOBDDs and a 256× improvement over WBDDs), 262,144 qubits for the BV and DJ benchmarks (a 2× improvement over CFLOBDDs and a 64× improvement over WBDDs), and 2,048 qubits for QFT (a 128× improvement over CFLOBDDs and a 2× improvement over WBDDs). These results demonstrate that, for certain application domains, WCFLOBDDs enable the handling of substantially larger problem instances than was previously feasible.

Finally, in Chapter 5, we presented Quasimodo, an open-source and easily

extensible framework for symbolic quantum simulation. QUASIMODO is a Python library that enables users to write quantum programs while leveraging symbolic simulation through a variety of underlying data structures. At present, QUASIMODO supports CFLOBDDs, BDDs, WBDDs, and WCFLOBDDs, and is designed to be readily extensible to additional representations. By cleanly separating the front-end quantum program from the simulation backend, QUASIMODO allows users to evaluate and compare different symbolic representations without modifying the program itself. This flexibility makes QUASIMODO a practical platform for experimenting with simulation strategies and for advancing research on succinct function representations in the context of quantum computing.

It should be noted that QUASIMODO strictly enforces a "physics-based" view of quantum-circuit simulation: a quantum circuit is specified as a sequence of gates acting on a given number of qubits, and a circuit is interpreted left-to-right, successively creating a representation of the state vector for each layer of gates. Moreover, the gates can only be created using a set of standard quantum gates—i.e., it does not support arbitrary unitary matrices as gates. For this reason, the performance of CFLOBDDs for quantum simulation via QUASIMODO is (sometimes) not as good as can be obtained when CFLOBDDs are used directly for quantum simulation by using simulation operations that lie outside the physics-based approach. In particular, Tab. 3.6 reports that CFLOBDDs can simulate 4096-qubit instances of Grover's Algorithm in 910 seconds, whereas the QUASIMODO paper Sistla et al. (2023) reports that QUASIMODO, when instantiated with CFLOBDDs, exceeds a 15-minute timeout threshold on 16-qubit instances. This substantial performance difference is due to (1) QUASIMODO carrying out a left-to-right simulation that generates the successive state vectors produced by the Grover circuit, whereas the results reported in Tab. 3.6 were obtained via the technique of repeated squaring of the (representation of) the unitary matrix for the Grover diffusion operator, as discussed in §2.3.1 and §3.8.2.6 and at the end of §3.9.2.2, and (2) the oracle, i.e., the gate encoding the hidden string, is constructed using operations beyond the standard gate-application primitives of a

quantum circuit.

Future Directions. The idea of hierarchy to capture repeated substructures for efficient representations opens several directions for future research. One promising direction for future research is the application of GCFLOBDDs to the domain of network verification. BDDs and their variants have long been used to represent network packets and capture network semantics through symbolic execution Yang and Lam (2015). More recently, Network Decision Diagrams (NDDs) Li et al. (2025) have been proposed as a hierarchical extension tailored specifically for network-verification tasks, enabling redundancy elimination. GCFLOBDDs provide an opportunity to further generalize these ideas of hierarchy and to explore their use for advancing symbolic techniques in network analysis.

Although this thesis primarily focuses on applying CFLOBDDs to quantum simulation and, to a lesser extent, to combinatorial circuits, these structures—along with GCFLOBDDs—have the potential to extend well beyond the domains explored here. One natural application is the representation of finite-state processes, where BDDs have traditionally been employed. Another promising direction is their use in model checking and probabilistic model checking, domains in which BDD-based techniques are already well established.

Model-checking algorithms typically involve iterative computations until a fixed point is reached. A similar pattern arises in Grover's algorithm, where CFLOBDDs demonstrate strong performance (in particular, see the discussion of repeated squaring toward the end of §3.9.2.2), suggesting their potential effectiveness in such settings. This connection points to a promising role for CFLOBDDs in model-checking tasks. However, as with BDDs, their effectiveness ultimately depends on how well the structure of the underlying transition system can be exploited by the representation.

Another research direction is the exploration of CFLOBDDs for linear-algebra

computations. BDDs have previously been applied to a range of linear-algebra operations beyond matrix multiplication, including Singular Value Decomposition (SVD), LU decomposition, and Cholesky decomposition. While we conducted preliminary investigations into adapting CFLOBDDs for this setting, the results were inconclusive. Many of the algorithms required for these operations do not naturally exhibit the type of regular structure that enables effective sharing within CFLOBDDs. One contributing factor lies in the differing computational paradigms of BDDs and CFLOBDDs. Algorithms built on BDDs often follow a depth-first strategy, resolving computations incrementally along individual paths from the root to the leaves. In contrast, CFLOBDD-based algorithms adopt a more breadth-oriented symbolic propagation, aggregating information across levels before resolving it at the top-most level. This mode of computation can limit opportunities for early simplification and thereby hinder scalability in linear-algebra workflows. Nevertheless, these observations do not preclude the possibility of successfully applying hierarchical decision diagrams to linear algebra. Future work could investigate alternative algorithmic strategies better aligned with the CFLOBDD computation model, as well as the use of GCFLOBDDs with custom grammars to capture structure specific to linear-algebra problems. Such efforts may reveal new pathways for leveraging hierarchy in large-scale symbolic numerical computation.

In Chapter 7, we examined how the choice of grammar in GCFLOBDDs influences the efficiency of function representations. Our results indicate that selecting an appropriate grammar can significantly improve performance. However, as the complexity of the target function increases, identifying a suitable grammar becomes increasingly challenging (similar to the difficulty in choosing the right variable ordering in BDDs). This observation naturally raises the question of whether grammars can be inferred automatically for a given problem. Developing techniques to learn or synthesize grammars that align with the structural properties of a function represents an important direction for future research. Such methods could reduce the reliance on manual design and enable GCFLOBDDs to be used more effectively in diverse

application domains.

In Chapters 3 and 4, we demonstrated how CFLOBDDs and WCFLOBDDs can be used to represent individual quantum states symbolically. In contrast, recent work Chen et al. (2023); Abdulla et al. (2025) employs tree automata to represent *sets* of quantum states, enabling the verification of quantum circuits over collections of states rather than a single configuration. An interesting direction for future research is to investigate whether CFLOBDDs can be extended with ideas from these automata-based approaches to support the symbolic representation of state sets. Such a fusion of approaches holds promise because the use of both hierarchical structure and linear structure in CFLOBDDs has advantages over the use of a strict hierarchical structure in tree automata, as was explored in our comparison of CFLOBDDs and TIDDs in Chapter 6. Such an integration could potentially lead to more scalable verification techniques for quantum circuits.

Appendix A: Details of Notation for CFLOBDDs and their Components

A few words are in order about the notation used in the pseudo-code:

- A Java-like semantics is assumed. For example, an object or field that is declared to be of type `InternalGrouping` is really a pointer to a piece of heap-allocated storage. A variable of type `InternalGrouping` is declared and initialized to a new `InternalGrouping` object of level k by the declaration

```
InternalGrouping g = new InternalGrouping(k)
```

- Procedures can return multiple objects by returning tuples of objects, where tupling is denoted by square brackets. For instance, if `f` is a procedure that returns a pair of `int`s—and, in particular, if `f(3)` returns a pair consisting of the values 4 and 5—then `int` variables `a` and `b` would be assigned 4 and 5 by the following initialized declaration:

```
int×int [a,b] = f(3)
```

- The indices of array elements start at 1.
- Arrays are allocated with an initial length (which is allowed to be 0); however, arrays are assumed to lengthen automatically to accommodate assignments at index positions beyond the current length.
- We assume that a call on the constructor `InternalGrouping(k)` returns an `InternalGrouping` in which the members have been initialized as follows:

```
level = k
AConnection = NULL
AReturnTuple = NULL
numberOfBConnections = 0
BConnections = new array[0] of Grouping
BReturnTuples = new array[0] of ReturnTuple
numberOfExits = 0
```

Similarly, we assume that a call on the constructor `CFLOBDD(g,vt)` returns a `CFLOBDD` in which the members have been initialized as follows:

```
grouping = g
valueTuple = vt
```

The class definitions of Fig. 3.2, as well as the algorithms for the core CFLOBDD operations make use of the following auxiliary classes:

- A `ReturnTuple` is a finite tuple of positive integers.
- A `PairTuple` is a sequence of ordered pairs.
- A `TripleTuple` is a sequence of ordered triples.
- A `ValueTuple` is a finite tuple of whatever values the multi-terminal CFLOBDD is defined over.

Appendix B: Lexicographic-Order Proposition of CFLOBDDs

Proposition B.1. 3.1 (LEXICOGRAPHIC-ORDER PROPOSITION). Let ex_C be the sequence of exit vertices of proto-CFLOBDD C. Let ex_L be the sequence of exit vertices reached by traversing C on each possible Boolean-variable-to-Boolean-value assignment, generated in lexicographic order of assignments. Let s be the subsequence of ex_L that retains just the leftmost occurrences of members of ex_L (arranged in order as they first appear in ex_L). Then $ex_C = s$.

Proof: We argue by induction over levels:

Base case: The proposition follows immediately for level-0 proto-CFLOBDDs.

Induction step: The induction hypothesis is that the proposition holds for every level-k proto-CFLOBDD.

Let C be an arbitrary level-k+1 proto-CFLOBDD, with s and ex_C as defined above. Without loss of generality, we will refer to the exit vertices by ordinal position; i.e., we will consider ex_C to be the sequence $[1, 2, \ldots, |ex_C|]$. Let C_A denote the A-connection of C, and let C_{B_n} denote C's n^{th} B-connection. Note that C_A and each of the C_{B_n} are level-k proto-CFLOBDDs, and hence, by the induction hypothesis, the proposition holds for them.

We argue by contradiction: Suppose, for the sake of argument, that the proposition does not hold for C, and that j is the leftmost exit vertex in ex_C for which the proposition is violated (i.e., $s(j) \neq j$). Let i be the exit vertex that appears in the jth position of s (i.e., $s(j) = i$). It must be that $j < i$.

Let α_j and α_i be the earliest assignments in lexicographic order (denoted by $\prec$) that lead to exit vertices j and i, respectively. Because i comes before j in s, it must be that $\alpha_i \prec \alpha_j$.

Let α_j^1 and α_j^2 denote the first and second halves of α_j, respectively; let α_i^1 and α_i^2 denote the first and second halves of α_i, respectively. Let $+$ denote the concatentation of assignments (e.g., $\alpha_j = \alpha_j^1 + \alpha_j^2$).

There are two cases to consider.

Case 1: $\alpha_i^1 = \alpha_j^1$ and $\alpha_i^2 \prec \alpha_j^2$.

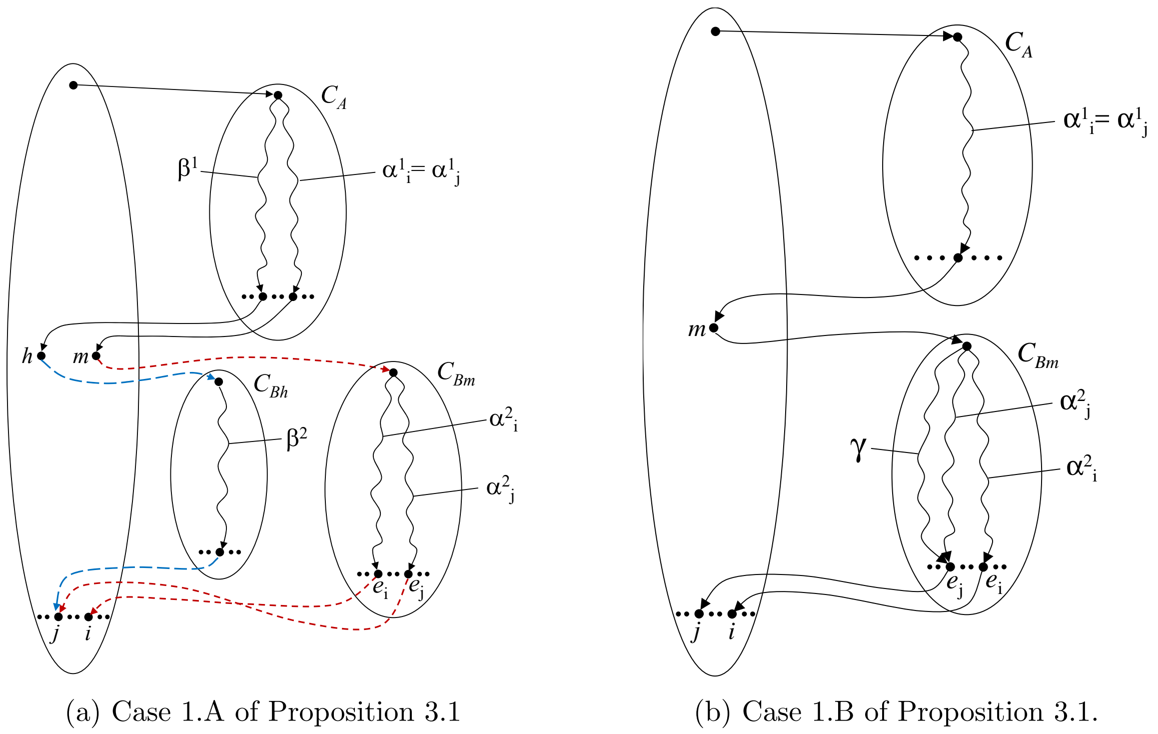
Because $\alpha_i^1 = \alpha_j^1$, the first halves of the matched path followed during the interpretations of assignments α_i and α_j through C_A are identical, and bring us to some middle vertex, say m, of C; both paths then proceed through C_{B_m}. Let e_i and e_j be the two exit vertices of C_{B_m} reached by following matched paths during the interpretations of α_i^2 and α_j^2, respectively. There are now two cases to consider:

Case 1.A: Suppose that $e_i < e_j$ in C_{B_m} (see Fig. B.1a). In this case, the return edges $e_i \to i$ and $e_j \to j$ "cross". By Structural Invariant 2b, this can only happen if

- There is a matched path corresponding to some assignment β^1 through C_A that leads to a middle vertex h, where $h < m$.
- There is a matched path from h corresponding to some assignment β^2 through C_{B_h} (where C_{B_h} could be C_{B_m}).
- There is a return edge from the exit vertex reached by β^2 in C_{B_h} to exit vertex j of C.

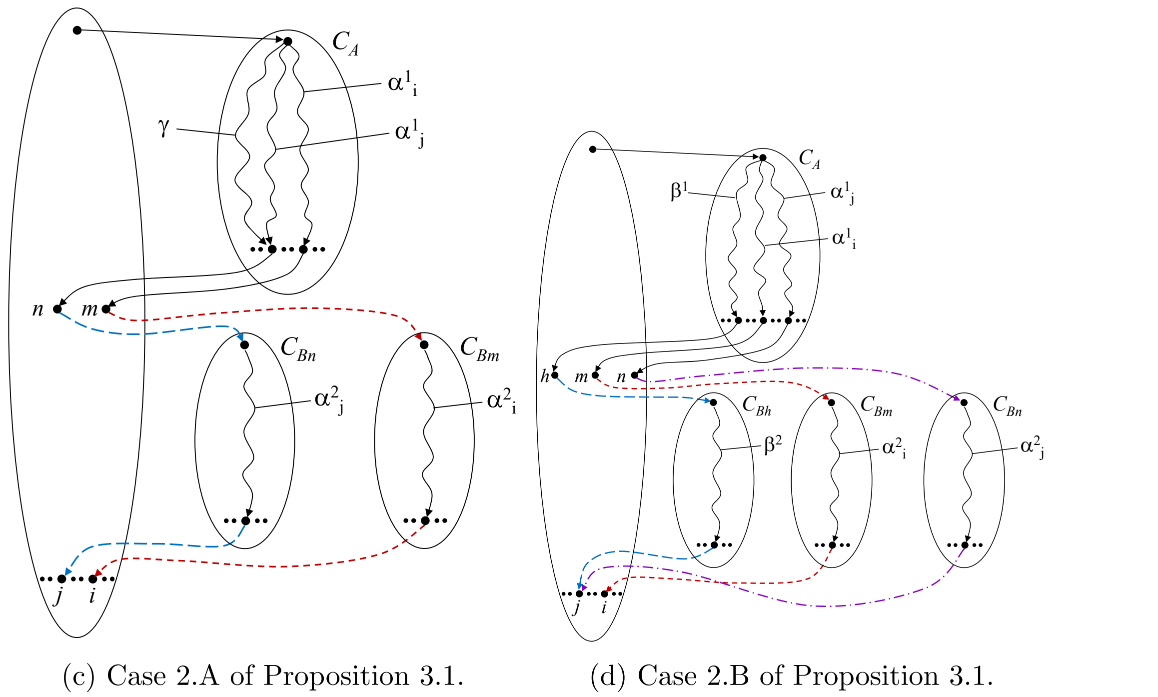
In this case, by the induction hypothesis applied to C_A, and the fact that $h < m$, it must be the case that we can choose β^1 so that $\beta^1 \prec \alpha_j^1$.

Consequently, $\beta^1 + \beta^2 \prec \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j.



(a) Case 1.A of Proposition 3.1

(b) Case 1.B of Proposition 3.1.



(c) Case 2.A of Proposition 3.1.

(d) Case 2.B of Proposition 3.1.

Figure B.1

Case 1.B: Suppose that $e_j < e_i$ in C_{B_m} (see Fig. B.1b). Because $\alpha_i^2 \prec \alpha_j^2$, the induction hypothesis applied to C_{B_m} implies that there must exist an assignment $\gamma \prec \alpha_i^2 \prec \alpha_j^2$ that leads to e_j. In this case, we have that $\alpha_j^1 + \gamma \prec \alpha_j^1 + \alpha_j^2$, which again contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j.

Case 2: $\alpha_i^1 \prec \alpha_j^1$.

Because $\alpha_i^1 \prec \alpha_j^1$, the first halves of the matched paths followed during the interpretations of assignments α_i and α_j through C_A bring us to two different middle vertices of C, say m and n, respectively. The two paths then proceed through C_{B_m} and C_{B_n} (where it could be the case that $C_{B_m} = C_{B_n}$), and return to i and j, respectively, where $j < i$. Again, there are two cases to consider:

Case 2.A: Suppose that $n < m$ (see Fig. B.1c.) The argument is similar to Case 1.B above: By Structural Invariant 1, $n < m$ means that the exit vertex reached by α_j^1 in C_A comes before the exit vertex reached by α_i^1 in C_A. By the induction hypothesis applied to C_A, there must exist an assignment $\gamma \prec \alpha_i^1 \prec \alpha_j^1$ that leads to the exit vertex reached by α_j^1 in C_A. In this case, we have that $\gamma + \alpha_j^2 \prec \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j.

Case 2.B: Suppose that $m < n$ (see Fig. B.1d.) The argument is similar to Case 1.A above: By Structural Invariant 2, we can only have $m < n$ and $j < i$ if

- There is a matched path corresponding to some assignment β^1 through C_A that leads to a middle vertex h, where $h < m$.
- There is a matched path from h corresponding to some assignment β^2 through C_{B_h} (where C_{B_h} could be C_{B_m} or C_{B_n}).
- There is a return edge from the exit vertex reached by β^2 in C_{B_h} to exit vertex j of C.

In this case, by the induction hypothesis applied to C_A, and the fact that $h < m < n$, it must be the case that we can choose β^1 so that $\beta^1 \prec \alpha_j^1$.

Consequently, $\beta^1 + \beta^2 \prec \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j.

In each of the cases above, we are able to derive a contradiction to the assumption that α_j is the least assignment in lexicographic order that leads to j. Thus, the supposition that the proposition does not hold for C cannot be true.

Appendix C: Proof of the Canonicalness of CFLOBDDs

To show that CFLOBDDs are a canonical representation of functions over Boolean arguments, we must establish that three properties hold:

1. Every level-k CFLOBDD represents a decision tree with 2^{2^k} leaves.
2. Every decision tree with 2^{2^k} leaves is represented by some level-k CFLOBDD.
3. No decision tree with 2^{2^k} leaves is represented by more than one level-k CFLOBDD (up to isomorphism).

As described earlier, following a matched path (of length $O(2^k)$) from the level-k entry vertex of a level-k CFLOBDD to a final value provides an interpretation of a Boolean assignment on 2^k variables. Thus, the CFLOBDD represents a decision tree with 2^{2^k} leaves (and Obligation 1 is satisfied).

To show that Obligation 2 holds, we describe a recursive procedure for constructing a level-k CFLOBDD from an arbitrary decision tree with 2^{2^k} leaves (i.e., of height 2^k). In essence, the construction shows how such a decision tree can be folded together to form a multi-terminal CFLOBDD.

The construction makes use of a set of auxiliary tables, one for each level, in which a unique representative for each class of equal proto-CFLOBDDs that arises is tabulated. We assume that the level-0 table is already seeded with a representative fork grouping and a representative don't-care grouping.

Construction 1. [Decision Tree to Multi-Terminal CFLOBDD]

1. *The leaves of the decision tree are partitioned into some number of equivalence classes e according to the values that label the leaves. The equivalence classes*

are numbered 1 to e according to the relative position of the first occurrence of a value in a left-to-right sweep over the leaves of the decision tree.

For Boolean-valued CFLOBDDs, when the procedure is applied at topmost level, there are at most two equivalence classes of leaves, for the values F and T. However, in general, when the procedure is applied recursively, more than two equivalence classes can arise.

For the general case of multi-terminal CFLOBDDs, the number of equivalence classes corresponds to the number of different values that label leaves of the decision tree.

2. *(**Base cases**) If $k = 0$ and $e = 1$, construct a CFLOBDD consisting of the representative don't-care grouping, with a value tuple that binds the exit vertex to the value that labels both leaves of the decision tree.*

   If $k = 0$ and $e = 2$, construct a CFLOBDD consisting of the representative fork grouping, with a value tuple that binds the two exit vertices to the first and second values, respectively, that label the leaves of the decision tree.

   If either condition applies, return the CFLOBDD so constructed as the result of this invocation; otherwise, continue on to the next step.

3. *Construct—via recursive applications of the procedure—$2^{2^{k-1}}$ level-k–1 multi-terminal CFLOBDDs for the $2^{2^{k-1}}$ decision trees of height 2^{k-1} in the lower half of the decision tree.*

   These are then partitioned into some number e' of equivalence classes of equal multi-terminal CFLOBDDs; a representative of each class is retained, and the others discarded. Each of the $2^{2^{k-1}}$ "leaves" of the upper half of the decision tree is labeled with the appropriate equivalence-class representative for the subtree of the lower half that begins there. These representatives serve as the "values" on the leaves of the upper half of the decision tree when the construction process is applied recursively to the upper half in step 4.

The equivalence-class representatives are also numbered 1 to e' according to the relative position of their first occurrence in a left-to-right sweep over the leaves of the upper half of the decision tree.

4. *Construct—via a recursive application of the procedure—a level-k–1 multi-terminal CFLOBDD for the upper half of the decision tree.*

5. *Construct a level-k multi-terminal proto-CFLOBDD from the level-k–1 multi-terminal CFLOBDDs created in steps 3 and 4. The level-k grouping is constructed as follows:*

   (a) *The A-connection points to the proto-CFLOBDD of the object constructed in step 4.*

   (b) *The middle vertices correspond to the equivalence classes formed in step 3, in the order $1 \ldots e'$.*

   (c) *The A-connection return tuple is the identity map back to the middle vertices (i.e., the tuple $[1..e']$).*

   (d) *The B-connections point to the proto-CFLOBDDs of the e' equivalence-class representatives constructed in step 3, in the order $1 \ldots e'$.*

   (e) *The exit vertices correspond to the initial equivalence classes described in step 1, in the order $1 \ldots e$.*

   (f) *The B-connection return tuples connect the exit vertices of the highest-level groupings of the equivalence-class representatives retained from step 3 to the exit vertices created in step 5e. In each of the equivalence-class representatives retained from step 3, the value tuple associates each exit vertex x with some value v, where $1 \leq v \leq e$; x is now connected to the exit vertex created in step 5e that is associated with the same value v.*

   (g) *Consult a table of all previously constructed level-k groupings to determine whether the grouping constructed by steps 5a–5f duplicate a previously constructed grouping. If so, discard the present grouping and switch to the*

previously constructed one; if not, enter the present grouping into the table.

6. *Return a multi-terminal CFLOBDD created from the proto-CFLOBDD constructed in step 5 by attaching a value tuple that connects (in order) the exit vertices of the proto-CFLOBDD to the e values from step 1.*

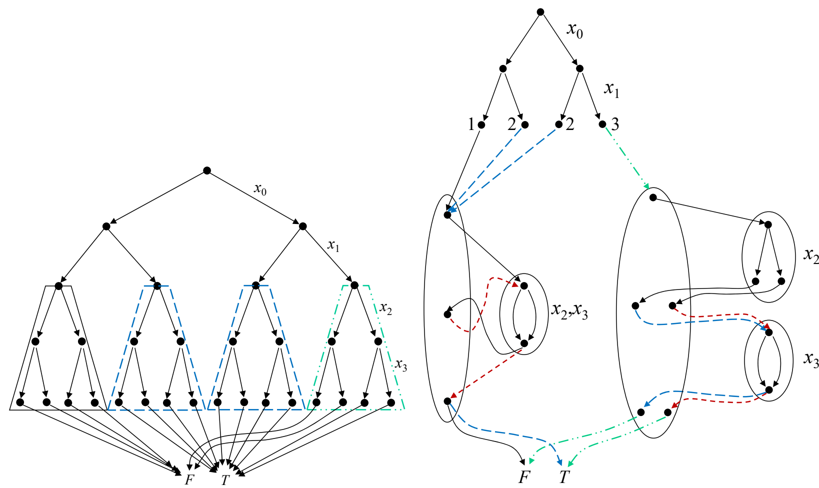
□

Fig. C.1a shows the decision tree for the function $\lambda x_0 x_1 x_2 x_3 . (x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. Fig. C.1b shows the state of things after step 3 of Construction 1. Note that even though the level-1 CFLOBDDs for the first three leaves of the top half of the decision tree have equal proto-CFLOBDDs,[1] the leftmost proto-CFLOBDD maps its exit vertex to F, whereas the exit vertex is mapped to T in the second and third proto-CFLOBDDs. Thus, in this case, the recursive call for the upper half of the decision tree (step 4) involves three equivalence classes of values.

It is not hard to see that the structures created by Construction 1 obey the structural invariants that are required of CFLOBDDs:

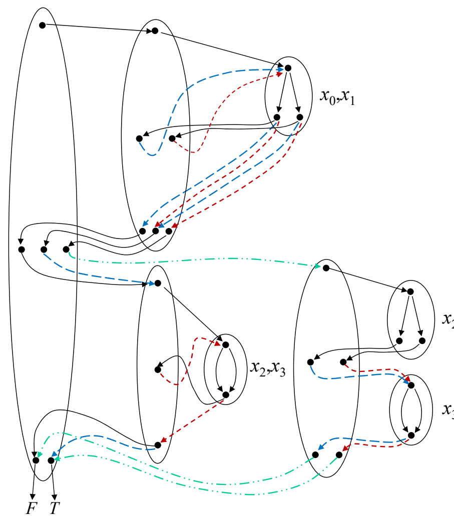
- Structural Invariant 1 holds because the A-connection return tuple created in step 5c of Construction 1 is the identity map.
- Structural Invariant 2 holds because in steps 1 and 3 of Construction 1, the equivalence classes are numbered in increasing order according to the relative position of a value's first occurrence in a left-to-right sweep. In particular, this order is preserved in the exit vertices of each grouping constructed during an invocation of Construction 1 (cf. step 5f), which ensures that the "compact extension" property of Structural Invariant 2b holds at each level of recursion in Construction 1.

[1] The equality of the proto-CFLOBDDs is detected in step 5g.



(a) Decision tree

(b) Hybrid of decision tree for x_0 and x_1, and CFLOBDDs for x_2 and x_3. The solid, dashed, and dashed-double-dotted edges from the four vertices labeled 1, 2, 2, and 3, respectively, correspond to the solid, dashed, and dashed-double-dotted trapezoids in (a).



(c) CFLOBDD (repeated from Fig. 3.3). For clarity, some of the level-0 groupings have been duplicated.

Figure C.1: Representations of the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$.

- Structural Invariant 3 holds because Construction 1 reuses the representative don't-care grouping and the representative fork grouping in step 2, and checks for the construction of duplicate groupings—and hence duplicate proto-CFLOBDDs—in step 5g.

- Structural Invariant 4 holds because of steps 3, 5d, and 5f. On recursive calls to Construction 1, step 3 partitions the CFLOBDDs constructed for the lower half of the decision tree into equivalence classes of CFLOBDD values (i.e., taking into account both the proto-CFLOBDDs and the value tuples associated with their exit vertices). Therefore, in steps 5d and 5f, duplicate B-connection/return-tuple pairs can never arise.

- Structural Invariant 5 holds because step 6 uses the proto-CFLOBDD constructed in step 5.

- Structural Invariant 6 holds because step 1 of Construction 1 constructs equivalence classes of values (ordered in increasing order according to the relative position of a value's first occurrence in a left-to-right sweep over the leaves of the decision tree).

Moreover, Construction 1 preserves interpretation under assignments: Suppose that C_T is the level-k CFLOBDD constructed by Construction 1 for decision tree T; it is easy to show by induction on k that for every assignment α on the 2^k Boolean variables $x_0, \ldots, x_{2^k-1}$, the value obtained from C_T by following the corresponding matched path from the entry vertex of C_T's highest-level grouping is the same as the value obtained for α from T. (The first half of α is used to follow a path through the A-connection of C_T, which was constructed from the top half of T. The second half of α is used to follow a path through one of the B-connections of C_T, which was constructed from an equivalence class of bottom-half subtrees of T; that equivalence class includes the subtree rooted at the vertex of T that is reached by following the first half of α.) Thus, every decision tree with 2^{2^k} leaves is represented by some level-k

CFLOBDD in which meaning (interpretation under assignments) has been preserved; consequently, Obligation 2 is satisfied.

We now come to Obligation 3 (no decision tree with 2^{2^k} leaves is represented by more than one level-k CFLOBDD). The way we prove this property is to define an unfolding process, called *Unfold*, that starts with a multi-terminal CFLOBDD and works in the opposite direction to Construction 1 to construct a decision tree; that is, *Unfold* (recursively) unfolds the A-connection, and then (recursively) unfolds each of the B-connections. For instance, for the example shown in Fig. C.1, *Unfold* would proceed from Fig. C.1c, to Fig. C.1b, and then to the decision tree for the function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$ shown in Fig. C.1a.

Unfold also preserves interpretation under assignments: Suppose that T_C is the decision tree constructed by *Unfold* for level-k CFLOBDD C; it is easy to show by induction on k that for every assignment α on the 2^k Boolean variables $x_0, \ldots, x_{2^k-1}$, the value obtained from C by following the corresponding matched path from the entry vertex of C's highest-level grouping is the same as the value obtained for α from T_C. (The first half of α is used to follow a path through the A-connection of C, which *Unfold* unfolds into the top half of T_C. The second half of α is used to follow a path through one of the B-connections of C, which *Unfold* unfolds into one or more instances of bottom-half subtrees of T_C; that set of bottom-half subtrees includes the subtree rooted at the vertex of T that is reached by following the first half of α.)

Obligation 3 is satisfied if we can show that, for every CFLOBDD C, Construction 1 applied to the decision tree produced by *Unfold*(C) yields a CFLOBDD that is isomorphic to C. To establish that this property holds, we will define two kinds of *traces*:

- A *Fold trace* records the steps of Construction 1:
  - At step 1 of Construction 1, the decision tree is appended to the trace.

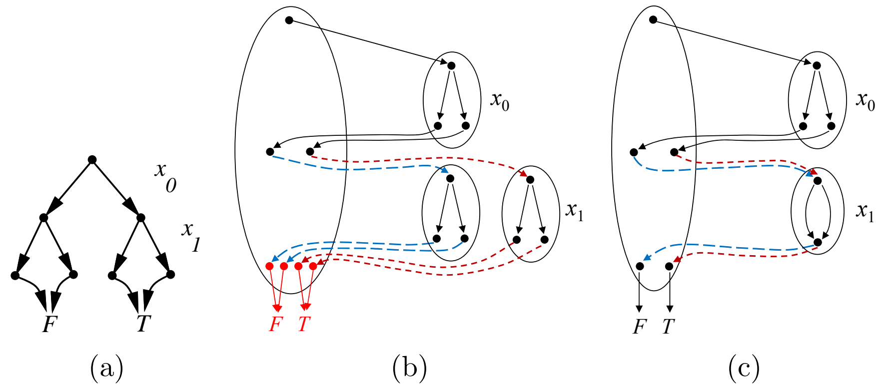


Figure C.2: (a) Decision tree for $\lambda x_0 x_1 . x_0$; (b) fully expanded form of the CFLOBDD; (c) CFLOBDD.

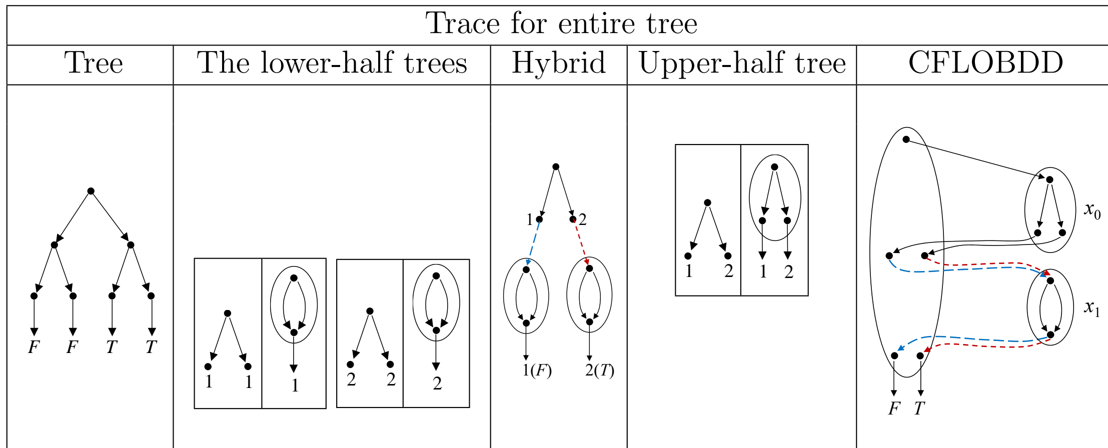


Figure C.3: The *Fold* trace generated by the application of Construction 1 to the decision tree shown in Fig. C.2a to create the CFLOBDD shown in Fig. C.2c.

- At the end of step 2 (if either of the conditions listed in step 2 holds), the level-0 CFLOBDD being returned is appended to the trace (and Construction 1 returns).

- During step 3, the trace is extended according to the actions carried out by the folding process as it is applied recursively to each of the lower-half decision trees. (For purposes of settling Obligation 3, we will assume that the lower-half decision trees are processed by Construction 1 in *left-to-right*

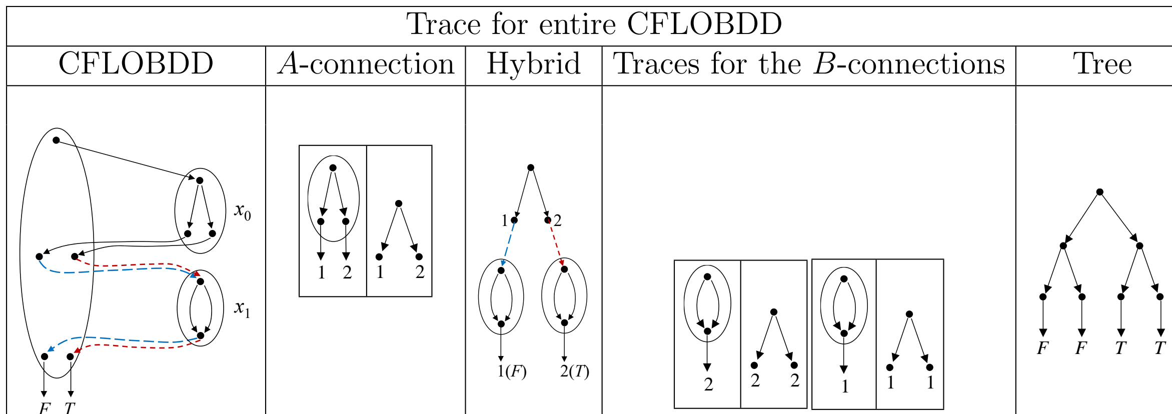


Figure C.4: The *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Fig. C.2c to create the decision tree shown in Fig. C.2a.

order.)

- At the end of step 3, a hybrid decision-tree/CFLOBDD object (à la Fig. C.1b) is appended to the trace.
- During step 4, the trace is extended according to the actions carried out by the folding process as it is applied recursively to the upper half of the decision tree.
- At the end of step 6, the CFLOBDD being returned is appended to the trace.

For instance, Fig. C.3 shows the *Fold* trace generated by the application of Construction 1 to the decision tree shown in Fig. C.2a to create the CFLOBDD shown in Fig. C.2c.

- An *Unfold trace* records the steps of *Unfold*(C):
  - CFLOBDD C is appended to the trace.
  - If C is a level-0 CFLOBDD, then a binary tree of height 1—with the leaves labeled according to C's value tuple—is appended to the trace (and the

Unfold algorithm returns).

– The trace is extended according to the actions carried out by *Unfold* as it is applied recursively to the A-connection of C.

– A hybrid decision-tree/CFLOBDD object (à la Fig. C.1b) is appended to the trace.

– The trace is extended according to the actions carried out by *Unfold* as it is applied recursively to instances of B-connections of C. (For purposes of settling Obligation 3, we will assume that *Unfold* processes a separate instance of a B-connection for each leaf of the hybrid object's upper-half decision tree, and that the B-connections are processed in *right-to-left* order of the upper-half decision tree's leaves.)

– Finally, the decision tree returned by *Unfold* is appended to the trace.

For instance, Fig. C.4 shows the *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Fig. C.2c to create the decision tree shown in Fig. C.2a.

Note how the *Unfold* trace shown in Fig. C.4 is the reversal of the *Fold* trace shown in Fig. C.3. We now argue that this property holds generally. (Technically, the argument given below in Proposition C.1 shows that each element of an *Unfold* trace is *isomorphic* to the corresponding object in the *Fold* trace, which suffices to imply that that Obligation 3 is satisfied, in the sense that a decision tree is represented by exactly one isomorphism class of CFLOBDDs.)

Proposition C.1. Suppose that C is a multi-terminal CFLOBDD, and that *Unfold*(C) results in *Unfold* trace UT and decision tree T_0. Let C' be the multi-terminal CFLOBDD produced by applying Construction 1 to T_0, and FT be the *Fold* trace produced during this process. Then

(i) FT is the reversal of UT.

(ii) C and C' are isomorphic.

Proof: Because C' appears at the end of FT, and C appears at the beginning of UT, clause (i) implies (ii). We show clause (i) by the following inductive argument:

Base case: The proposition is trivially true of level-0 CFLOBDDs. Given any pair of values v_1 and v_2 (such as F and T), there are exactly four possible level-0 CFLOBDDs: two constructed using a don't-care grouping—one in which the exit vertex is mapped to v_1, and one in which it is mapped to v_2—and two constructed using a fork grouping—one in which the two exit vertices are mapped to v_1 and v_2, respectively, and one in which they are mapped to v_2 and v_1, respectively. These unfold to the four decision trees that have $2^{2^0} = 2$ leaves and leaf-labels drawn from $\{v_1, v_2\}$, and the application of Construction 1 to these decision trees yields the same level-0 CFLOBDD that we started with. (See step 2 of Construction 1.) Consequently, the *Fold* trace FT and the *Unfold* trace UT are reversals of each other.

Induction step: The induction hypothesis is that the proposition holds for every level-k multi-terminal CFLOBDD. We need to argue that the proposition extends to level-k+1 multi-terminal CFLOBDDs.

First, note that the induction hypothesis implies that each decision tree with 2^{2^k} leaves is represented by exactly one level-k CFLOBDD isomorphism class. We will refer to this as the *corollary to the induction hypothesis*.

Unfold trace UT can be divided into five segments:

(u1) C itself

(u2) the *Unfold* trace for C's A-connection

(u3) a hybrid decision-tree/CFLOBDD object (call this object D)

(u4) the *Unfold* trace for C's B-connections

(u5) T_0.

Fold trace FT can also be divided into five segments:

(f1) T_0

(f2) the *Fold* trace for T_0's lower-half trees

(f3) a hybrid decision-tree/CFLOBDD object (call this object D')

(f4) the *Fold* trace for T_0's upper-half

(f5) C'.

Because both (f1) and (u5) are T_0, (u5) is obviously equal to (f1). Our goal, therefore, is to show that

- (u2) is the reversal of (f4);
- (u3) is equal to (f3);
- (u4) is the reversal of (f2); and
- (u1) is equal to (f5).

(u3) is equal to (f3) Consider the hybrid decision-tree/CFLOBDD object D obtained after *Unfold* has finished unfolding C's A-connection.[2] The upper part of D (the decision-tree part) came from the recursive invocation of *Unfold*, which produced a decision tree for the first half of the Boolean variables, in

[2]The A-connection is actually a proto-CFLOBDD, whereas *Unfold* works on multi-terminal CFLOBDDs. However, the A-connection return tuple (with the indices of the middle vertices as the value space) serves as the value tuple whenever we wish to consider the A-connection as a multi-terminal CFLOBDD.

which each leaf is labeled with the index of a middle vertex from the level-$k{+}1$ grouping of C (e.g., see Fig. C.1b).

As a consequence of Prop. 3.1, together with the fact that *Unfold* preserves interpretation under assignments, the relative position of the first occurrence of a label in a left-to-right sweep over the leaves of this decision tree reflects the order of the level-$k{+}1$ grouping's middle vertices.[3] However, each middle vertex has an associated B-connection, and by Structural Invariants 2, 4, and 6, the middle vertices can be thought of as representatives for a set of pairwise non-equal CFLOBDDs (that themselves represent lower-half decision trees).

Fold trace FT also has a hybrid decision-tree/CFLOBDD object, namely D'. The crucial point is that the action of partitioning T_0's lower-half CFLOBDDs that is carried out in step 3 of Construction 1 also results in a labeling of each leaf of the upper-half's decision tree with a representative of an equivalence class of CFLOBDDs that represent the lower half of the decision tree starting at that point.

By the corollary to the induction hypothesis, the 2^{2^k} bottom-half trees of T_0 are represented uniquely (up to isomorphism) by the respective CFLOBDDs in D'. Similarly, by the corollary to the induction hypothesis, the 2^{2^k} CFLOBDDs used as labels in D represent uniquely (up to isomorphism) the respective bottom-half trees of T_0. Thus, the labelings on D and D' must be isomorphic.

(u2) is the reversal of (f4); (u4) is the reversal of (f2) Given the observation that D and D' are isomorphic, these properties follow in a straightforward fashion from the inductive hypothesis (applied to the A-connection and the B-connections of C).

[3]This step is where the argument would break down if we attempt to apply the same argument to Fig. 3.6a: In that case, the labels on the leaves of D, in left-to-right order, would be 2 and 1—whereas the sequence of middle vertices in Fig. 3.6a is [1,2].

(u1) is equal to (f5) Because (u2) is the reversal of (f4) and (u4) is the reversal of (f2), we know that the level-k proto-CFLOBDDs out of which the level-k+1 grouping of C' is constructed are isomorphic to the respective level-k proto-CFLOBDDs that make up the A-connection and B-connections of C.

We already argued that steps 5 and 6 of Construction 1 lead to CFLOBDDs that obey the six structural invariants required of CFLOBDDs. Moreover, there is only one way for Construction 1 to construct the level-k+1 grouping of C' so that Structural Invariants 2, 3, and 4 are satisfied. Therefore, C is isomorphic to C'.

Consequently, FT is the reversal of UT, as was to be shown.

In summary, we have now shown that Obligations 1, 2, and 3 are all satisifed. These properties imply that, for a given ordering of Boolean variables, if two level-k CFLOBDDs C_1 and C_2 represent the same decision tree with 2^{2^k} leaves, then C_1 and C_2 are isomorphic—i.e., CFLOBDDs are a canonical representation of functions over Boolean arguments:

Theorem 3.2**.** (Canonicity). If C_1 and C_2 are level-k CFLOBDDs for the same Boolean function over 2^k Boolean variables, and C_1 and C_2 use the same variable ordering, then C_1 and C_2 are isomorphic.

Appendix D: Pair Product Canonicity Proof for CFLOBDDs

We prove that the grouping `g` constructed during a call on `PairProduct` meets Structural Invariant 4—and hence it is permissible to call `RepresentativeGrouping(g)` in line [38] of Alg. 13.

In particular, suppose that (i) B_1 and B_1' are B-connections of a grouping g_1 (with associated return tuples rt_1 and rt_1', respectively), (ii) B_2 and B_2' are B-connections of a grouping g_2 (with associated return tuples rt_2 and rt_2', respectively), and (iii) at least one of the following two conditions hold:

1. $\langle B_1, rt_1 \rangle \neq \langle B_1', rt_1' \rangle$
2. $\langle B_2, rt_2 \rangle \neq \langle B_2', rt_2' \rangle$

In addition, suppose that the recursive calls on `PairProduct` produce

$$[D, pt] = \texttt{PairProduct}(B_1, B_2) \qquad \text{and} \qquad [D', pt'] = \texttt{PairProduct}(B_1', B_2'),$$

Let rt and rt' be the return tuples that the outer calls on `PairProduct` in line [22] of Alg. 13 create for D and D': pt, rt_1, and rt_2 are used to create rt; pt', rt_1', and rt_2' are used to create rt'.

The question that we need to answer is whether it is ever possible for both $D = D'$ and $rt = rt'$ to hold. This question is of concern because the hypothesized condition would violate Structural Invariant 4: if the condition were to hold, then the first entry of the pair returned by `PairProduct` would not be a well-formed proto-CFLOBDD. The following proposition shows that, in fact, this situation cannot ever occur:

Proposition D.1. The first entry of the pair returned by `PairProduct` is always a well-formed proto-CFLOBDD.

Proof: We argue by induction:

Base case: When g_1 and g_2 are level-0 groupings, there are four cases to consider. In each case, it is immediate from lines [2]–[5] of Alg. 12 that the first entry of the pair returned by `PairProduct` is a well-formed proto-CFLOBDD.

Induction step: The induction hypothesis is that the first entry of the pair returned by `PairProduct` is a well-formed proto-CFLOBDD whenever the arguments to `PairProduct` are level-k proto-CFLOBDDs.

Let g_1 and g_2 be two arbitrary well-formed level-k+1 proto-CFLOBDDs. We argue by contradiction: suppose, for the sake of argument, that D, D', rt, and rt' are as defined above, and that both $D = D'$ and $rt = rt'$ hold.

- By the inductive hypothesis, we know that D and D' are each well-formed level-k proto-CFLOBDDs. In particular, we can think of D and rt as corresponding to a decision tree T_0, labeled with the exit vertices of g to which the decision tree's leaves are mapped. However, because of the search that is carried out in lines [24]–[36] of `PairProduct` (Alg. 13), each exit vertex of g corresponds to a unique pair, $\langle c_1, c_2 \rangle$, where c_1 and c_2 are exit vertices of g_1 and g_2, respectively. Thus, a leaf in T_0 can be thought of as being labeled with a pair $\langle c_1, c_2 \rangle$.

  Furthermore, because $D = D'$ and $rt = rt'$, D' and rt' also correspond to decision tree T_0.

- When T_0 is considered to be the decision tree associated with D and rt, we can read off (a) the decision tree that corresponds to B_1 with exit vertices of g_1 labeling the leaves (call this tree T_1), and (b) the decision tree that corresponds

to B_2 with exit vertices of g_2 labeling the leaves (T_2). Similarly, when T_0 is considered to be the decision tree associated with D' and rt', we can read off (c) the decision tree that corresponds to B'_1 with exit vertices of g_1 labeling the leaves (T'_1), and (d) the decision tree that corresponds to B'_2 with exit vertices of g_2 labeling the leaves (T'_2). (We use the first entry of each $\langle c_1, c_2 \rangle$ pair for B_1 and B'_1, and the second entry of each $\langle c_1, c_2 \rangle$ pair for B_2 and B'_2.) This process gives us four trees, T_1, T'_1, T_2, and T'_2, where—from the supposition that $D = D'$ and $rt = rt'$—we must have $T_1 = T'_1$ and $T_2 = T'_2$.

- By assumption, g_1 and g_2 are well-formed proto-CFLOBDDs; thus, by Structural Invariant 2, all return tuples for the B-connections of g_1 and g_2 must represent 1-to-1 maps. Moreover, B_1, B_2, B'_1, and B'_2 are also well-formed proto-CFLOBDDs, which means that, in g_1, B_1 together with rt_1 must be the unique representative of occurrences of T_1 in g_1's decision tree, while B'_1 together with rt'_1 must be the unique representative of occurrences of T'_1. Similarly, in g_2, B_2 together with rt_2 must be the unique representative of occurrences of T_2 in g_2's decision tree, while B'_2 together with rt'_2 must be the unique representative of occurrences of T'_2.

  Therefore, in g_1, we have

  - $B_1 = B'_1$ and $rt_1 = rt'_1$,

  while in g_2, we have

  - $B_2 = B'_2$ and $rt_2 = rt'_2$.

  However, these conditions are a violation of Structural Invariant 4, which, in turn, contradicts the assumption that g_1 and g_2 are well-formed level-k+1 proto-CFLOBDDs. Consequently, the assumption that $D = D'$ and $rt = rt'$ cannot be true.

Algorithm 48: TernaryApplyAndReduce

```
   Input: CFLOBDDs n1, n2, n3, Op op
   Output: CFLOBDD n = op(n1, n2, n3)
 1 begin
 2   assert(n1.grouping.level == n2.grouping.level && n2.grouping.level
       == n3.grouping.level);
     // Perform triple cross product
 3   Grouping×TripleTuple [g,tt] = TripleProduct(n1.grouping,
       n2.grouping, n3.grouping);
     // Create tuple of "leaf" values
 4   ValueTuple deducedValueTuple = [op(n1.valueTuple[i1],
       n2.ValueTuple[i2], n3.ValueTuple[i3]) : [i1,i2,i3]∈ tt];
     // Collapse duplicate leaf values, folding to the left
 5   Tuple×Tuple [inducedValueTuple,inducedReturnTuple] =
       CollapseClassesLeftmost(deducedValueTuple);
     // Perform corresponding reduction on g, folding g's exit
        vertices w.r.t.  inducedReductionTuple
 6   Grouping g' = Reduce(g, inducedReductionTuple);
 7   return RepresentativeCFLOBDD(g', inducedValueTuple);
 8 end
```

Appendix E: Additional Operations on CFLOBDDs

E.1 Ternary Operations on CFLOBDDs

This section discusses how ternary operations (i.e., three-argument operations) on CFLOBDDs are performed. Algs. 48 and 49 present the two algorithms needed to implement ternary operations on multi-terminal CFLOBDDs. As in §3.6.3, we assume that the `CFLOBDD` or `Grouping` arguments of the operations described below are objects whose highest-level groupings are all at the same level.

- The operation `TernaryApplyAndReduce` given in Alg. 48 is very much like the operation

`BinaryApplyAndReduce` of Alg. 11, except that it starts with a call on `TripleProduct` instead of `PairProduct` (line [3]).

- The operation `TripleProduct`, which is given in Alg. 49, is very much like the operation `PairProduct` of Alg. 12, except that `TripleProduct` has a third `Grouping` argument, and performs a three-way—rather than two-way—cross product of the three `Grouping` arguments: `g1`, `g2`, and `g3`. `TripleProduct` returns the proto-CFLOBDD `g` formed in this way, as well as a descriptor of the exit vertices of `g` in terms of triples of exit vertices of the highest-level groupings of `g1`, `g2`, and `g3`.

  (By an argument similar to the one given for `PairProduct`, it is possible to show that the grouping `g` constructed during a call on `TripleProduct` is always a well-formed proto-CFLOBDD—and hence it is permissible to call `RepresentativeGrouping(g)` in line [55] of Alg. 49.)

- `TernaryApplyAndReduce` then uses the triples describing the exit vertices to determine the tuple of leaf values that should be associated with the exit vertices (i.e., a tentative value tuple) (line [4]).

- Finally, `TernaryApplyAndReduce` proceeds in the same manner as `BinaryApplyAndReduce`:

  - Two tuples that describe the collapsing of duplicate leaf values—assuming folding to the left—are created via a call to `CollapseClassesLeftmost` (line [5]).

  - The corresponding reduction is performed on `Grouping g`, by calling `Reduce` to fold `g`'s exit vertices with respect to variable `inducedReductionTuple` (one of the tuples returned by the call on `CollapseClassesLeftmost`) (line [6]).

Lastly, in the case of Boolean-valued CFLOBDDs, there are 256 possible ternary operations, corresponding to the 256 possible three-argument truth tables ($2 \times 2 \times 2$ matrices with Boolean entries). All 256 possible ternary operations are special cases of `TernaryApplyAndReduce`; these can be performed by passing `TernaryApplyAndReduce` an appropriate value for argument `op` (i.e., some $2 \times 2 \times 2$ Boolean matrix.

One of the 256 ternary operations is the operation called ITE Brace et al. (1991) (for "If-Then-Else"), which is defined as follows:

$$\text{ITE}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c).$$

Appendix §F shows how the ternary ITE operation can be used to implement all 16 of the binary operations on Boolean-valued CFLOBDDs Brace et al. (1991).

E.2 Restrict

We now discuss the restriction operation. Given a value v to which variable x_i is to be bound (e.g., by giving either the assignment $[x_i \mapsto T]$ or the assignment $[x_i \mapsto F]$), the Restrict operation applies the assignment to the CFLOBDD that represents a function f, and returns the CFLOBDD that represents $f|_{x_i=v}$. Algs. 51 and 52 gives pseudo-code for the algorithm. At each level, the algorithm checks if index i belongs to the *A-Connection* or *B-Connections* at every level, and calls Restrict recursively, as appropriate, on lower levels with an adjusted index value i. The rest of the groupings are kept as is, except that some groupings are eliminated, and the positions of the remaining ones shifts (Alg. 52, lines [3] and [15]).

E.3 Existential Quantification

For a CFLOBDD that represents a Boolean function f, existential quantification with respect to the Boolean variable at index i yields $f|_{x_i==T} \vee f|_{x_i==F}$. This

operation can be implemented using two calls to Restrict, followed by a final call on `BinaryApplyAndReduce` to perform the “or.”

Truth Table	Defining Expression	Definition Using ITE
b: F T a F: F F a T: F F	$\lambda a, b.F$	$\lambda a, b.F$
b: F T a F: F F a T: F T	$\lambda a, b.a \wedge b$	$\lambda a, b.\text{ITE}(a, b, F)$
b: F T a F: F F a T: T F	$\lambda a, b.a \wedge \neg b$	$\lambda a, b.\text{ITE}(a, \neg b, F)$
b: F T a F: F F a T: T T	$\lambda a, b.a$	$\lambda a, b.a$
b: F T a F: F T a T: F F	$\lambda a, b.\neg a \wedge b$	$\lambda a, b.\text{ITE}(a, F, b)$
b: F T a F: F T a T: F T	$\lambda a, b.b$	$\lambda a, b.b$
b: F T a F: F T a T: T F	$\lambda a, b.a \oplus b$	$\lambda a, b.\text{ITE}(a, \neg b, b)$
b: F T a F: F T a T: T T	$\lambda a, b.a \vee b$	$\lambda a, b.\text{ITE}(a, T, b)$
b: F T a F: T F a T: F F	$\lambda a, b.\neg(a \vee b)$	$\lambda a, b.\text{ITE}(a, F, \neg b)$
b: F T a F: T F a T: F T	$\lambda a, b.\neg(a \oplus b)$	$\lambda a, b.\text{ITE}(a, b, \neg b)$
b: F T a F: T F a T: T F	$\lambda a, b.\neg b$	$\lambda a, b.\text{ITE}(b, F, T)$
b: F T a F: T F a T: T T	$\lambda a, b.a \vee \neg b$	$\lambda a, b.\text{ITE}(a, T, \neg b)$
b: F T a F: T T a T: F F	$\lambda a, b.\neg a$	$\lambda a, b.\text{ITE}(a, F, T)$
b: F T a F: T T a T: F T	$\lambda a, b.\neg a \vee b$	$\lambda a, b.\text{ITE}(a, b, T)$
b: F T a F: T T a T: T F	$\lambda a, b.\neg(a \wedge b)$	$\lambda a, b.\text{ITE}(a, \neg b, T)$
b: F T a F: T T a T: T T	$\lambda a, b.T$	$\lambda a, b.T$

Figure E.1

Algorithm 49: TripleProduct

```
   Input: Groupings g1, g2, g3
   Output: Grouping g: product of g1, g2, g3; TripleTuple ttAns: tuple of
           triples of exit vertices
 1 begin
 2    if g1, g2, g3 are all no-distinction proto-CFLOBDDs then
 3       return [g1, [[1,1,1]]];
 4    end
 5    if g1 and g2 no-distinction proto-CFLOBDDs then
 6       return [g3, [[1,1,k]: k ∈ [1..g3.numberOfExits]];
 7    end
 8    if g1 and g3 no-distinction proto-CFLOBDDs then
 9       return [g2, [[1,k,1]: k ∈ [1..g2.numberOfExits]];
10    end
11    if g2 and g3 no-distinction proto-CFLOBDDs then
12       return [g1, [[k,1,1]: k ∈ [1..g1.numberOfExits]];
13    end
14    if g1 is a no-distinction proto-CFLOBDD then
15       Grouping×PairTuple [g,pt] = PairProduct(g2,g3);
16       return [g,[[1,j,k]: [j,k]∈ pt]];
17    end
18    if g2 is a no-distinction proto-CFLOBDD then
19       Grouping×PairTuple [g,pt] = PairProduct(g1,g3);
20       return [g,[[j,1,k]: [j,k]∈ pt]];
21    end
22    if g3 is a no-distinction proto-CFLOBDD then
23       Grouping×PairTuple [g,pt] = PairProduct(g1,g2);
24       return [g,[[j,k,1]: [j,k]∈ pt]];
25    end
26    if g1,g2,g3 are all fork groupings then
27       return [g1, [[1,1,1],[2,2,2]];
28    end
      // Combine the A-Connections
29    Grouping×TripleTuple [gA, ttA] = TripleProduct(g1.AConnection,
        g2.AConnection, g3.AConnection);
30    InternalGrouping g = new InternalGrouping(g1.level);
31    g.AConnection = gA;
32    g.AReturnTuple = [1..|ttA|];  // Represents the middle vertices
33    g.numberOfBConnections = |ttA|;
```

Algorithm 50: TripleProduct Contd.

```
35
      // Combine the B-connections, but only for triples in ttA
      // Descriptor of triples of exit vertices
36    Tuple ttAns = [];
      // Create a B-Connection for each middle vertex
37    for j ← 1 to |ttA| do
38        Grouping×TripleTuple [gB,ttB] =
           TripleProduct(g1.BConnections[ttA(j)(1)],
           g2.BConnections[ttA(j)(2)], g3.BConnections[ttA(j)(3)])];
39        g.BConnections[j] = gB;
40        g.BReturnTuples[j] = [];
41        for i ← 1 to |ttB| do
42            c1 = g1.BReturnTuples[ttA(j)(1)](ttB(i)(1));
43            c2 = g2.BReturnTuples[ttA(j)(1)](ttB(i)(2));
44            c3 = g3.BReturnTuples[ttA(j)(1)](ttB(i)(3));
              // Not a new exit vertex of g
45            if [c1,c2,c3] ∈ ttAns then
46                index = the k such that ttAns(k) == [c1,c2,c3];
47                g.BReturnTuples[j] = g.BReturnTuples[j] || index;
48            else
49                g.numberOfExits = g.numberOfExits + 1;
50                g.BReturnTuples[j] = g.BReturnTuples[j] ||
                   g.numberOfExits;
51                ttAns = ttAns || [c1,c2,c3];
52            end
53        end
54    end
55    return [RepresentativeGrouping(g), ttAns];
56 end
```

Algorithm 51: RestrictCFLOBDD

Input: CFLOBDD c representing f, int i – index, bool v – T/F

Output: CFLOBDD c' representing $f' = f|_{x_i=v}$

```
1 begin
2     Grouping×ReturnTuple [g, rt] = RestrictGrouping(c.grouping, i, v);
3     return RepresentativeCFLOBDD(g, [g.valueTuple[rt[i]] | i ∈ [1..|rt|]]);
4 end
```

Algorithm 52: RestrictGrouping

```
   Input: Grouping g, int i, bool v
   Output: Grouping×ReturnTuple [g', grt]
 1 begin
 2   if g == ForkGrouping then
 3     return [DontCareGrouping, (v == False) ? [1] : [2]];
 4   end
 5   if g == DontCareGrouping || g ==
      NoDistinctionProtoCFLOBDD(g.level) then
 6     return [g, [1]];
 7   end
 8   InternalGrouping g' = new InternalGrouping(g.level);
 9   if i < 2**(g.level-1) then        // i falls in AConnection range
10     Grouping×ReturnTuple [aa, apt] =
        RestrictGrouping(g.AConnection, i, v);
11     g'.AConnection = aa;
12     g'.AReturnTuple = [1..|apt|];
13     g'.numberOfBConnections = |apt|;
14     for j ← 1 to |apt| do
15       g'.BConnections[j] = g.BConnections[apt[j]];
         // Fill g'.BReturnTuples[j] from
            g.BReturnTuples[apt[j]]; populate grt
         // Keep track of number of exits of g'
16     end
17   else                              // i falls in BConnections range
18     for j ← 1 to g.numberOfBConnections do
19       Grouping×ReturnTuple [bb, bpt] =
          RestrictGrouping(g.BConnections[j], i-(1 << (g.level-1)), v);
20       g'.BConnections[j] = bb;
         // Fill g'.BReturnTuples[j] from bpt; populate grt
         // Keep track of number of exits of g'
21     end
22     g'.AConnection = g.AConnection;
23     g'.AReturnTuples = [1..|distinct g'.BConnections|];
24     g'.numberOfBConnections = |distinct g'.BConnections|;
25   end
26   g'.numberOfExits = number of exits tracked so far;
27   return [RepresentativeGrouping(g'), grt];
28 end
```

Appendix F: Boolean Operations via ITE

The ternary operation ITE Brace et al. (1991) (for "If-Then-Else"), is defined as follows:

$$\text{ITE}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c).$$

Fig. E.1 shows how the ternary ITE operation can be used to implement all 16 of the binary operations on Boolean-valued CFLOBDDs Brace et al. (1991).

Appendix G: Kronecker Product with CFLOBDDs

This section presents and discusses pseudo-code for the variant of Kronecker product sketched in §3.6.5.1. Given CFLOBDDs for matrices W and V, with the interleaved-variable orderings $x \bowtie y$ and $w \bowtie z$, respectively, the goal is to create a CFLOBDD for $W \otimes V$ with variable ordering $(x||w) \bowtie (y||z)$.[1]

Algorithm 53: Kronecker Product

Input: CFLOBDDs n1, n2 with variable ordering of n1: $x \bowtie y$ and n2: $w \bowtie z$

Output: CFLOBDD n = n1 $\otimes$ n2 with variable ordering of n: $(x||w) \bowtie (y||z)$

```
1 begin
      // Create a CFLOBDD of size level(n1) + 1 with n1 as the
         AConnection
2     CFLOBDD g1 =
       RepresentativeCFLOBDD(ShiftToAConnection(n1.grouping),
       n1.valueTuple);
      // Create a CFLOBDD of size level(n2) + 1 with n2 as the
         BConnection
3     CFLOBDD g2 =
       RepresentativeCFLOBDD(ShiftToBConnection(n2.grouping),
       n2.valueTuple);
4     CFLOBDD n = BinaryApplyAndReduce(g1, g2, (op)Times);
5     return n;
6 end
```

Fig. 3.11a shows a level-k CFLOBDD for some array W, where W's value tuple is $[w_0, w_1, w_2]$; Fig. 3.11b shows a level-k CFLOBDD for some array V, where V's value tuple is $[v_0, v_1, v_2]$. (W and V could have been embedded into level-k+1 CFLOBDDs; for the sake of clarity, we have not depicted such structures.) Fig. 3.11c shows the level-k+1 CFLOBDD that would be constructed by Alg. 53 before any

[1] $\bowtie$ denotes the interleaving of two sequences of variables; $||$ denotes concatenation.

Algorithm 54: ShiftToAConnection

```
   Input: Grouping g
   Output: Grouping g' such that AConnection of g' = g
 1 begin
 2     InternalGrouping g' = new InternalGrouping(g.level + 1);
 3     g'.AConnection = g;
 4     g'.AReturnTuple = [1..|g.numberOfExits|];
 5     g'.numberOfBConnections = |g.numberOfExits|;
 6     for j ← 1 to g.numberOfExits do
 7         g'.BConnection[j] = NoDistinctionProtoCFLOBDD(g.level);
 8         g'.BReturnTuples[j] = [j];
 9     end
10     g'.numberOfExits = |g.numberOfExits|;
11     return RepresentativeGrouping(g');
12 end
```

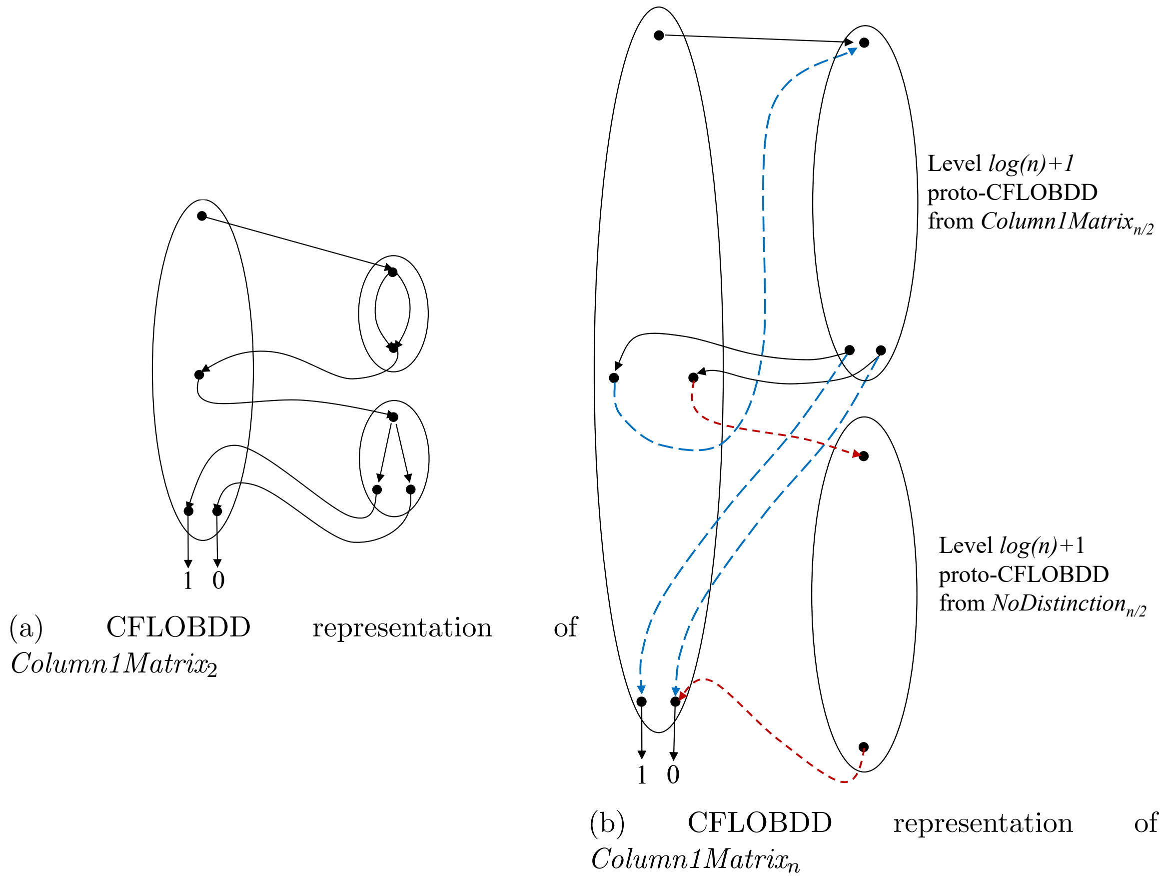
Algorithm 55: ShiftToBConnection

```
   Input: Grouping g
   Output: Grouping g' such that BConnection of g' = g
 1 begin
 2     InternalGrouping g' = new InternalGrouping(g.level + 1);
 3     g'.AConnection = NoDistinctionProtoCFLOBDD(g.level);
 4     g'.AReturnTuple = [1];
 5     g'.numberOfBConnections = 1;
 6     g'.BConnection[1] = g;
 7     g'.numberOfExits = |g.numberOfExits|;
 8     return RepresentativeGrouping(g');
 9 end
```

collapsing of the value tuple via `Reduce` in the call to `BinaryApplyAndReduce` in line [4].

Under the variable order $(x||w) \bowtie (y||z)$, as we work through the CFLOBDD shown in Fig. 3.11c for a given assignment, the values of the first 2^k Boolean variables lead us to a middle vertex of the level-k+1 grouping. This path will be continued according to the values of the next 2^k variables. Call these two paths p_A and p_B, respectively. Under the interleaved-variable ordering, p_A takes us to a particular block of the matrix that Fig. 3.11c represents, and p_B takes us to a particular element of that block.

The path p_A uses the 2^k variables in $x \bowtie y$, and thus can also be thought of as taking us to an exit vertex e_w that in matrix W is associated with some terminal value w_i. The path p_B uses the 2^k variables in $w \bowtie z$, and thus can be thought of as taking us to an exit vertex e_v that in matrix V is associated with some terminal value v_j. In the CFLOBDD shown in Fig. 3.11c, the terminal value at the end of the path $p_A||p_B$ is $w_i v_j$. This value is exactly what is required of the matrix $W \otimes V$. After `Reduce` is called on Fig. 3.11c, by the canonicity property, the multi-terminal CFLOBDD that results must be the unique representation of $W \otimes V$ under the variable ordering $(x||w) \bowtie (y||z)$.



(a) CFLOBDD representation of $Column1Matrix_2$

(b) CFLOBDD representation of $Column1Matrix_n$

Figure H.1: (a) Base case; (b) the recursive structure for general n.

Appendix H: Efficient Construction of $\boldsymbol{Column1Matrix_n}$ with CFLOBDDs

$Column1Matrix_n$ is a square matrix of size $2^n \times 2^n$ in which the first column is filled with 1s, and all other entries are 0.

$$Column1Matrix_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}_{2^n \times 2^n}$$

Algorithm 56: Column1Matrix

```
1  Algorithm ColumnMatrixCFLOBDD(l)
     Input: int l - level of the CFLOBDD = (log n) + 1, where 2n = the
            number of variables
     Output: CFLOBDD c representing Column1Matrix_n
2    begin
3      Grouping g = Column1MatrixGrouping(l);
4      return RepresentativeCFLOBDD(g, [1,0]);
5    end
6  end
1  SubRoutine ColumnMatrixGrouping(l)
     Input: int l - level of the CFLOBDD = (log n) + 1, where 2n = the
            number of variables
     Output: Grouping g representing proto - Column1Matrix_n
2    begin
3      InternalGrouping g = new InternalGrouping(l);
4      if l == 1 then
5        g.AConnection = ForkGrouping;
6        g.AReturnTuples = [1,2];
7        g.numberOfBConnections = 2;
8        g.BConnection[1] = DontCareGrouping;
9        g.ReturnTuples[1] = [1];
10       g.BConnection[2] = DontCareGrouping;
11       g.BReturnTuples[2] = [2];
12     else
13       Grouping g' = ColumnMatrixGrouping(l-1);
14       g.AConnection = g';
15       g.AReturnTuples = [1,2];
16       g.numberOfBConnections = 2;
17       g.BConnection[1] = g';
18       g.BReturnTuples[1] = [1,2];
19       g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
20       g.BReturnTuples[2] = [2];
21     end
22     g.numberOfExits = 2;
23     return RepresentativeGrouping(g);
24   end
25 end
```

Column1Matrix$_n$ can be recursively defined in terms of the matrices *Column1Matrix*$_{n/2}$ (of size $2^{n/2} \times 2^{n/2}$) and $O_{n/2}$ (the all-zero matrix of size $2^n \times 2^n$).

$$Column1Matrix_n = \begin{cases} \begin{bmatrix} Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \vdots & \vdots & \ddots & \vdots \\ Column1Matrix_{n/2} & O_{n/2} & \cdots & O_{n/2} \end{bmatrix}_{2^n \times 2^n} & \\ \quad = Column1Matrix_{n/2} \otimes Column1Matrix_{n/2} & n > 1 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & n = 1 \end{cases}$$

Base Case. The CFLOBDD representation for the base case of $n = 1$, *Column1Matrix*$_1$, is shown in Fig. H.1a. The base-case matrix requires two Boolean variables x_0 and y_0: x_0 specifies the row, and y_0 specifies the column; hence, the CFLOBDD that represents *Column1Matrix*$_1$ has two levels, 0 and 1. The rows of *Column1Matrix*$_1$ are identical, so the *A-Connection* grouping at level 1 (for x_0) is a *DontCareGrouping*. In contrast, the columns of *Column1Matrix*$_1$ are not identical, so the *B-Connection* grouping at level 1 (for y_0) is a *ForkGrouping*. See Fig. H.1a and lines [5]–[11] of `ColumnMatrixGrouping` in Alg. 56. Note that the left exit vertex of the level-1 proto-CFLOBDD represents the first-column entries of the matrix (which are to have the value 1), and the right exit vertex represents the matrix entries that are to have the value 0.

General Case ($n > 1$). The steps to create *Column1Matrix*$_n$, for $n > 1$, are shown in lines [13]–[20] of `ColumnMatrixGrouping` in Alg. 56. The number of levels in the CFLOBDD that represents *Column1Matrix*$_n$ is $\log n + 2$, to provide for the needed $2n$ Boolean variables. (The leaves are at level 0, so the outermost level is $l = \log n + 1$.) The *A-Connection* of the level-l grouping represents a function involving

the most-significant $\frac{n}{2}$ row and $\frac{n}{2}$ column variables. From the recursive definition of $\mathit{Column1Matrix}_n = \mathit{Column1Matrix}_{n/2} \otimes \mathit{Column1Matrix}_{n/2}$, the function for the first $\frac{n}{2}$ variables is obtained via a recursive call on $\mathit{Column1Matrix}_n$ with half of the variables. Hence, the *A-Connection* grouping at level-l is a proto-CFLOBDD that represents $\mathit{Column1Matrix}_{n/2}$ (Fig. H.1b and line [14] in Alg. 56). The number of exits of this proto-CFLOBDD is two (for which the invariant is maintained that the left exit vertex of the level-l–1 proto-CFLOBDD is associated with 1, and the right exit vertex is associated with 0). The grouping at level l therefore has two *B-Connection* groupings, the first one being a second use of the proto-CFLOBDD for $\mathit{Column1Matrix}_{n/2}$ (Fig. H.1b and line [17] in Alg. 56). The second *B-Connection* grouping at level-l is a proto-CFLOBDD for $\text{NoDistinction}_{n/2}$, representing $O_{n/2}$. This recursive structure is shown in Fig. H.1b.

At top level, the level-l grouping has two exit vertices—the first maps to the value 1 and the second maps to 0.

Appendix I: Algorithm for Constructing the CNOT Matrix using CFLOBDDs

Pseudo-code for the algorithm for constructing the CFLOBDD that represents the Controlled-NOT matrix is given in Algs. 57–61. The algorithm for the special-case construction of CNOT_n discussed in §3.8.1.5 is given in Alg. 62.

Algorithm 57: Controlled-NOT matrix

```
1  Algorithm CNOTCFLOBDD(n, i, j)
     Input: int n, where 2n = #Variables of the CFLOBDD, int i -
            control-bit; int j - controlled-bit
     Output: CFLOBDD c representing CNOT(n, i, j)
2    begin
3      Grouping g = CNOTGrouping(log n + 1, i, j);
4      return RepresentativeCFLOBDD(g, [1,0]);
5    end
6  end
1  SubRoutine CNOTGrouping(l, i, j)
     Input: int l - grouping level, int i - control-bit; int j - controlled-bit
     Output: Grouping g representing CNOT(l, i, j) with n = 2^l bits
2    begin
3      if l == 2 then                                         // Base Case
4        InternalGrouping g = new InternalGrouping(2);
5        InternalGrouping g' = new InternalGrouping(1);   // Level 1
6        g'.AConnection = ForkGrouping;
7        g'.AReturnTuple = [1,2];
8        g'.numberOfBConnections = 2;
9        g'.BConnection[1] = ForkGrouping;
10       g'.BReturnTuples[1] = [1,2];
11       g'.BConnection[2] = ForkGrouping;
12       g'.BReturnTuples[2] = [2,3];
13       g'.numberOfExits = 3;
14       g.AConnection = g';
15       g.AReturnTuple = [1,2,3];
16       g.numberOfBConnections = 3;
17       g.BConnection[1] = IdentityMatrixGrouping(1);
18       g.BReturnTuples[1] = [1,2];
19       g.BConnection[2] = NoDistinctionProtoCFLOBDD(1);
20       g.BReturnTuples[2] = [2];
21       g.BConnection[3] = IdentityMatrixGrouping(1);
22       g.BReturnTuples[3] = [2,1];
23       g.numberOfExits = 2;
24       return RepresentativeGrouping(g);
25     end
```

Algorithm 58: CNOT contd.

```
26
27
28        if i and j fall in A-connection range then                // Case 1
29            InternalGrouping g = new InternalGrouping(l + 1);
30            g.AConnection = CNOTGrouping(l-1, i, j);
31            g.AReturnTuple = [1,2];
32            g.numberOfBConnections = 2;
33            g.BConnection[1] = IdentityMatrixGrouping(g.level - 1);
34            g.BReturnTuples[1] = [1,2];
35            g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
36            g.BReturnTuples[2] = [2];
37            g.numberOfExits = 2;
38            return RepresentativeGrouping(g);
39        end
40        if i and j fall in B-connection range then                // Case 2
41            InternalGrouping g = new InternalGrouping(l + 1);
42            g.AConnection = IdentityMatrixGrouping(g.level - 1);
43            g.AReturnTuple = [1,2];
44            g.numberOfBConnections = 2;
45            g.BConnection[1] = CNOTGrouping(l-1, i', j');
               // i' = i - 2^(l-1),  j' = j - 2^(l-1)
46            g.BReturnTuples[1] = [1,2];
47            g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
48            g.BReturnTuples[2] = [2];
49            g.numberOfExits = 2;
50            return RepresentativeGrouping(g);
51        end
```

Algorithm 59: CNOT contd.

```
52
53
54      if i in A-connection range and j in B-connection range then
         // Case 3
55          InternalGrouping g = new InternalGrouping(l + 1);
56          g.AConnection = CNOTGrouping(l-1, i, -1);
57          g.AReturnTuple = [1,2,3];
58          g.numberOfBConnections = 3;
59          g.BConnection[1] = IdentityMatrixGrouping(g.level-1);
60          g.BReturnTuples[1] = [1,2];
61          g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
62          g.BReturnTuples[2] = [2];
63          g.BConnection[3] = CNOTGrouping(l-1, -1, j');
             // j' = j - 2^(l-1)
64          g.BReturnTuples[3] = [2,1];
65          g.numberOfExits = 2;
66          return RepresentativeGrouping(g);
67      end
68      if i in A-connection range but j is not in this range then // Case
         4
69          InternalGrouping g = new InternalGrouping(l + 1);
70          g.AConnection = CNOTGrouping(l-1, i, -1);
71          g.AReturnTuple = [1,2,3];
72          g.numberOfBConnections = 3;
73          g.BConnection[1] = IdentityMatrixGrouping(g.level-1);
74          g.BReturnTuples[1] = [1,2];
75          g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
76          g.BReturnTuples[2] = [2];
77          g.BConnection[3] = IdentityMatrixGrouping(g.level-1);
78          g.BReturnTuples[3] = [3,2] ;
79          g.numberOfExits = 3;
80          return RepresentativeGrouping(g);
81      end
```

Algorithm 60: CNOT contd.

```
82
83
84        if i in B-connection range but j is not in this range then // Case
           5
85            InternalGrouping g = new InternalGrouping(l + 1);
86            g.AConnection = IdentityMatrixGrouping(g.level-1);
87            g.AReturnTuple = [1,2];
88            g.numberOfBConnections = 2;
89            g.BConnection[1] = CNOTGrouping(l-1, i', -1); // i' = i - 2^{l-1}
90            g.BReturnTuples[1] = [1,2,3];
91            g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
92            g.BReturnTuples[2] = [2];
93            g.numberOfExits = 3;
94            return RepresentativeGrouping(g);
95        end
96        if j in A-connection range but i is not in this range then // Case
           6
97            InternalGrouping g = new InternalGrouping(l + 1);
98            g.AConnection = CNOTGrouping(l-1, -1, j);
99            g.AReturnTuple = [1,2];
100           g.numberOfBConnections = 2;
101           g.BConnection[1] = NoDistinctionProtoCFLOBDD(g.level-1);
102           g.BReturnTuples[1] = [1,2];
103           g.BConnection[2] = IdentityMatrixGrouping(g.level - 1);
104           g.BReturnTuples[2] = [1];
105           g.numberOfExits = 2;
106           return RepresentativeGrouping(g);
107       end
```

Algorithm 61: CNOT contd.

```
108
109
110         if j in B-connection range but i is not in this range then // Case
             7
111             InternalGrouping g = new InternalGrouping(l + 1);
112             g.AConnection = IdentityMatrixGrouping(g.level-1);
113             g.AReturnTuple = [1,2];
114             g.numberOfBConnections = 2;
115             g.BConnection[1] = CNOTGrouping(l-1, -1, j');
                 // j' = j - 2^(l-1)
116             g.BReturnTuples[1] = [1,2];
117             g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
118             g.BReturnTuples[2] = [1];
119             g.numberOfExits = 2;
120             return RepresentativeGrouping(g);
121         end
122     end
123 end
```

Algorithm 62: Algorithm for constructing $CNOT_n$

```
1  Algorithm CNOTInterleavedMatrixCFLOBDD(l)
     Input: int l - level of the CFLOBDD = log 2n + 1, where 2n =
            number of bits
     Output: The CFLOBDD that represents CNOT_n
2    begin
3      Grouping g = CNOTInterleavedMatrixGrouping(l);
4      return RepresentativeCFLOBDD(g, [1,0]);
5    end
6  end
1  SubRoutine CNOTInterleavedMatrixGrouping(l)
     Input: int l - level of the CFLOBDD = log n, where 2n = number of
            bits
     Output: Grouping g representing the proto-CFLOBDD for CNOT_n
2    begin
3      InternalGrouping g = new InternalGrouping(l);
4      if l == 2 then
5        return CNOTGrouping(2, 1, 2); // The base case is CNOT_2
6      else
7        Grouping g' = CNOTInterleavedMatrixGrouping(l-1);
8        g.AConnection = g';
9        g.AReturnTuple = [1,2];
10       g.numberOfBConnections = 2;
11       g.BConnection[1] = g';
12       g.BReturnTuples[1] = [1,2];
13       g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
14       g.BReturnTuples[2] = [2];
15     end
16     g.numberOfExits = 2;
17     return RepresentativeGrouping(g);
18   end
19 end
```

Appendix J: Construction of Additional Quantum Gates

In this section, we discuss the construction of two quantum gates: the Controlled Phase gate and the Swap gate.

J.1 Controlled-Phase Gate

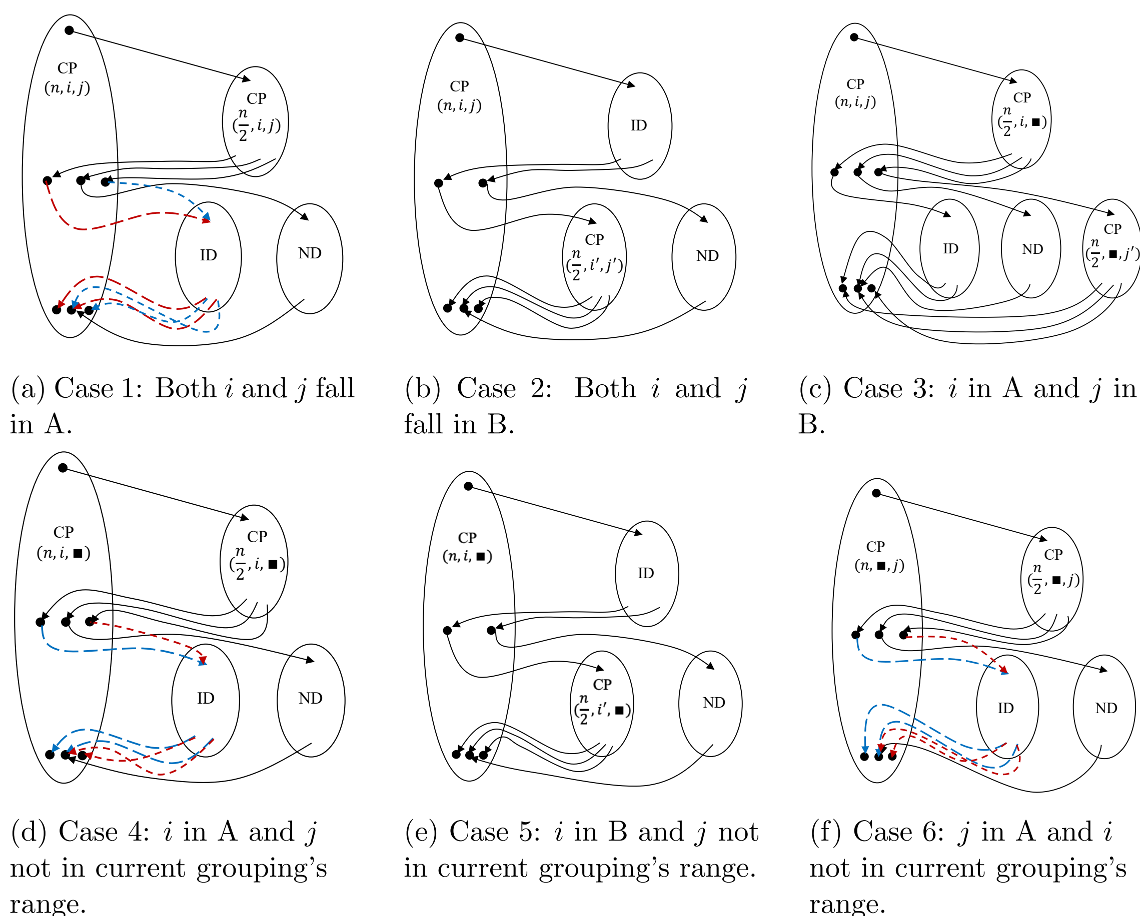
A Controlled-Phase gate (CP) for two qubits with angle θ has the following matrix:

$$CP(\theta) = \begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix} \end{array}$$

More generally, the controlled-phase gate with control qubit i and controlled qubit j adds the phase angle θ to the jth qubit when both qubits have the value 1. The construction of the CFLOBDD that represents a CP gate's matrix for n qubits is similar to the construction of the CFLOBDD for the $CNOT$ matrix. We do not give pseudo-code, but Figs. J.1 and J.2 depict the different cases of the construction of the proto-CFLOBDD for CP. The CFLOBDD for a CP gate attaches the three terminal values $[1, 0, e^{i\theta}]$.

	Role	Significance of exit vertex		
		1	2	3
Proto-CFLOBDD	$CP(n, i, j)$	on-path	off-path	phase-path
	$CP(n, i, ■)$	on-path	off-path	phase-path
	$CP(n, ■, j)$	on-path	off-path	phase-path
	ID called from middle vertex 1	on-path	off-path	N/A
	ID called from middle vertex 3	phase-path	off-path	N/A
CFLOBDD	Top level	1	0	$e^{i\theta}$

(J.1)



(a) Case 1: Both i and j fall in A.

(b) Case 2: Both i and j fall in B.

(c) Case 3: i in A and j in B.

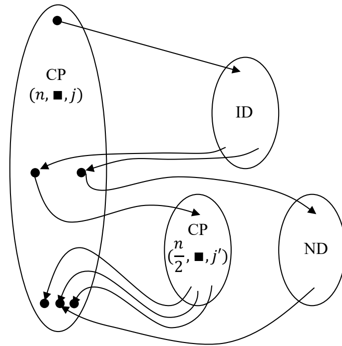
(d) Case 4: i in A and j not in current grouping's range.

(e) Case 5: i in B and j not in current grouping's range.

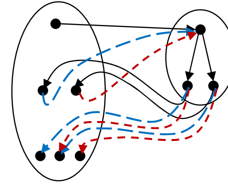
(f) Case 6: j in A and i not in current grouping's range.

Figure J.1: The different cases of the *CP* construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`, and ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). *CP* takes 4 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, where $0 \leq i < j < n$; and θ (which is only used at top level in the CFLOBDD's value tuple). i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range. Figure (h) shows the base case at level 1; the same proto-CFLOBDD is used for both $CP(n, 1, \blacksquare)$ and $CP(n, \blacksquare, 1)$. Continued in Fig. J.2

CP takes 4 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, where $0 \leq i < j < n$; and the phase θ. The key to understanding Figs. J.1 and J.2 is that the construction maintains the



(a) Case 7: j in B and i not in current grouping's range.



(b) Base case: the same proto-CFLOBDD is used for both $CP(n, 1, \blacksquare)$ (to interpret the control-bit) and $CP(n, \blacksquare, 1)$ (to interpret the controlled-bit).

Figure J.2: The different cases of the *CP* construction. Continued from Fig. J.1.

invariant shown in Eqn. (J.1) on the exit vertices of the different kinds of groupings.

Here, "on-path" means that the exit occurs on a matched-path that can be continued to the top-level terminal value 1; "off-path" means that it will only be used to reach the top-level terminal value 0; and "phase-path" means that the exit occurs on a matched-path that can be continued by `IdentityMatrixGroupings` to the top-level terminal values θ and 0.

J.2 Swap Gate

A Swap gate is a matrix that swaps two bits (or variables). The Swap matrix for 2 bits (i.e., for 2 input variables and 2 output variables) is

$$
\mathit{SwapGate} = \begin{array}{c} \\ 00 \\ 01 \\ 10 \\ 11 \end{array}
\begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \tag{J.2}
$$

A matrix entry is 1 in exactly the positions where swapping the bits of the row index yields the bits of the column index. The matrix can be divided into four quadrants that have different values:

$$
\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.
$$

This observation comes in handy in the construction of the generalized Swap matrix for higher numbers of bits n.

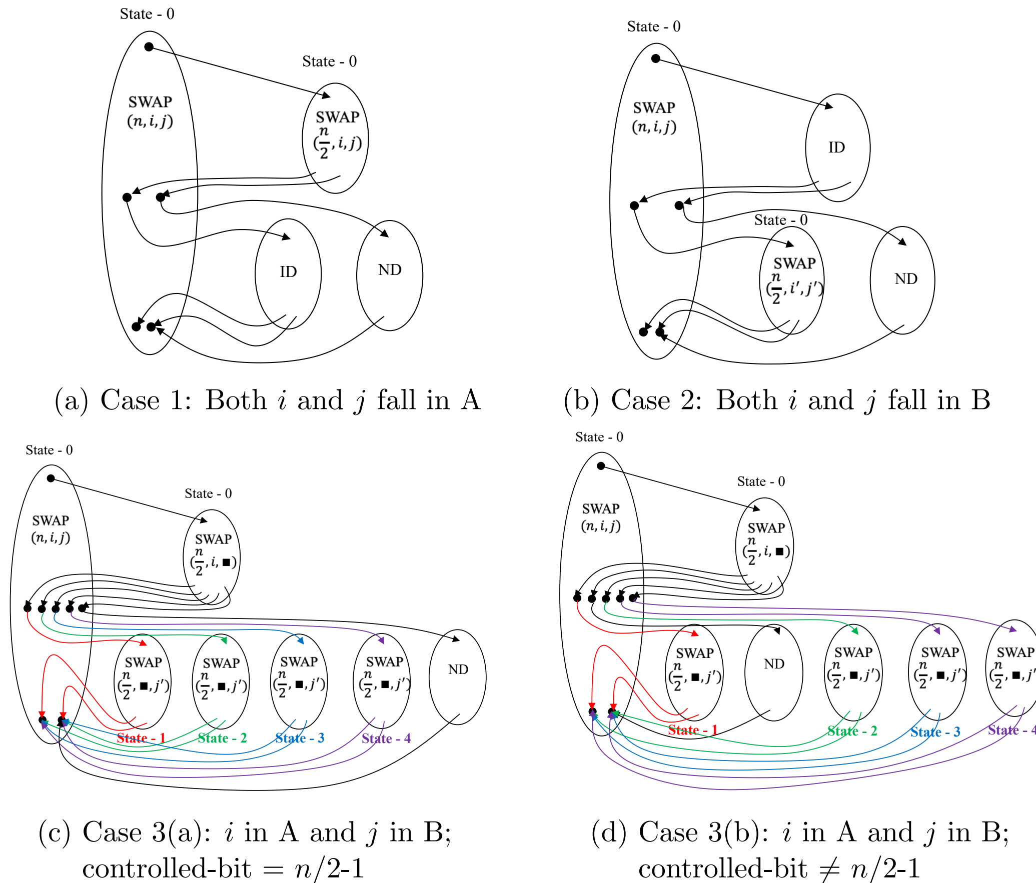
The construction of the *Swap* matrix is similar to the construction of the *CNOT* matrix. Again, we do not give pseudo-code, but give a graphical depiction of the different cases of the construction. The CFLOBDD for a *Swap* gate always has $[1, 0]$ as the value tuple for the terminal values. Figs. J.3, J.4, J.5, J.6 and J.7 depict the different cases involved in the construction of the proto-CFLOBDD for *Swap*, given n and the bits i and j to swap (where $i < j$). The construction also uses an additional parameter, called "*State*," which takes the values $\{0..4\}$. *State* is initially 0, and remains 0 until the Boolean variable corresponding to bit i is encountered. At this point, the construction creates representations of sub-matrices, where *State* takes the values 1, 2, 3, and 4, corresponding to the different 2×2 sub-matrices discussed above. This situation is depicted in the base case shown in Fig. J.6d. These states are propagated further through the proto-CFLOBDD (see Fig. J.4a, Fig. J.4b, Fig. J.5c, Fig. J.5d, Fig. J.6b, and Fig. J.6c) until the base cases that interpret the controlled-bit are encountered (see Fig. J.7a, Fig. J.7b, Fig. J.7c, and Fig. J.7d).

The key to understanding Figs. J.3, J.4, J.5, J.6 and J.7 is that the construction maintains the invariant shown in Eqn. (J.3) on the exit vertices of the different kinds of groupings. ("On-path" means that the exit occurs on a matched-path that can be

continued to the top-level terminal value 1; "off-path" means that it will only be used to reach the top-level terminal value 0; and "*cd-bit*" abbreviates "controlled-bit.")

	Role	Significance of exit vertex				
		1	2	3	4	5
Proto-	$SWAP(n,i,j)$	on-path	off-path	N/A	N/A	N/A
CFLOBDD	$SWAP(n,i,\blacksquare)$, *cd-bit* $= \frac{n}{2}$-1	*State* = 1	*State* = 2	*State* = 3	*State* = 4	off-path
	$SWAP(n,i,\blacksquare)$, *cd-bit* $\neq \frac{n}{2}$-1	*State* = 1	off-path	*State* = 2	*State* = 3	*State* = 4
	$SWAP(n,\blacksquare,j)$, *State* = 1	on-path	off-path	N/A	N/A	N/A
	$SWAP(n,\blacksquare,j)$, *State* = 2	off-path	on-path	N/A	N/A	N/A
	$SWAP(n,\blacksquare,j)$, *State* = 3	off-path	on-path	N/A	N/A	N/A
	$SWAP(n,\blacksquare,j)$, *State* = 4	off-path	on-path	N/A	N/A	N/A
	ID, called from *State* = 1	*State* = 1	off-path	N/A	N/A	N/A
	ID, called from *State* = 2	*State* = 2	off-path	N/A	N/A	N/A
	ID, called from *State* = 3	*State* = 3	off-path	N/A	N/A	N/A
	ID, called from *State* = 4	*State* = 4	off-path	N/A	N/A	N/A
CFLOBDD	Top level	1	0	N/A	N/A	N/A

(J.3)



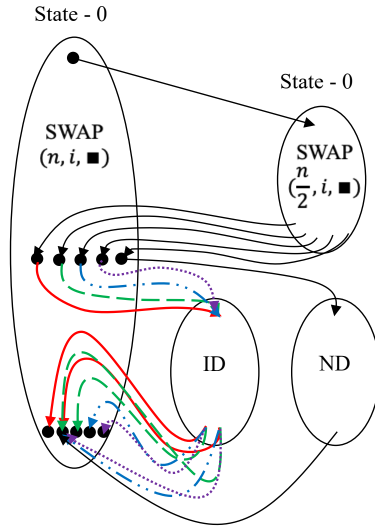
(a) Case 1: Both i and j fall in A

(b) Case 2: Both i and j fall in B

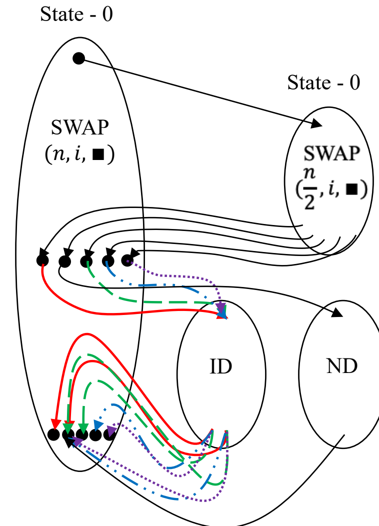
(c) Case 3(a): i in A and j in B; controlled-bit $= n/2$-1

(d) Case 3(b): i in A and j in B; controlled-bit $\neq n/2$-1

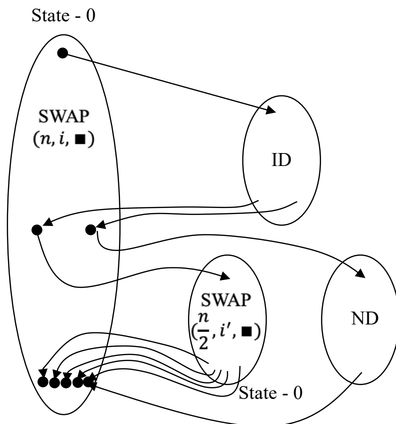
Figure J.3: The different cases of the $SWAP$ matrix construction. The text in each grouping denotes the function represented by the grouping. ID denotes `IdentityMatrixGrouping`, and ND denotes a `NoDistinctionProtoCFLOBDD` (used here for an all-zero matrix). $SWAP$ takes 4 arguments: n for the number of bits (number of variables will be $2n$) in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, and $State \in \{0, 1, 2, 3, 4\}$ to indicate the current mode of the construction. i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range.



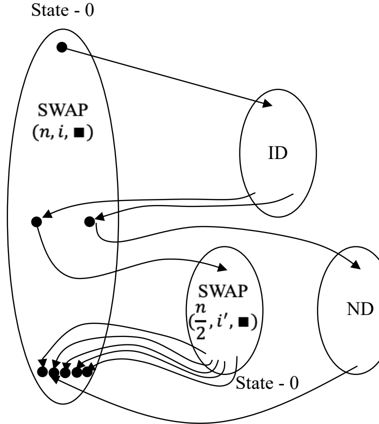
(a) Case 4(a): i in A and j not in current range; controlled-bit $= n/2$-1



(b) Case 4(b): i in A and j not in current range; controlled-bit $\neq n/2$-1

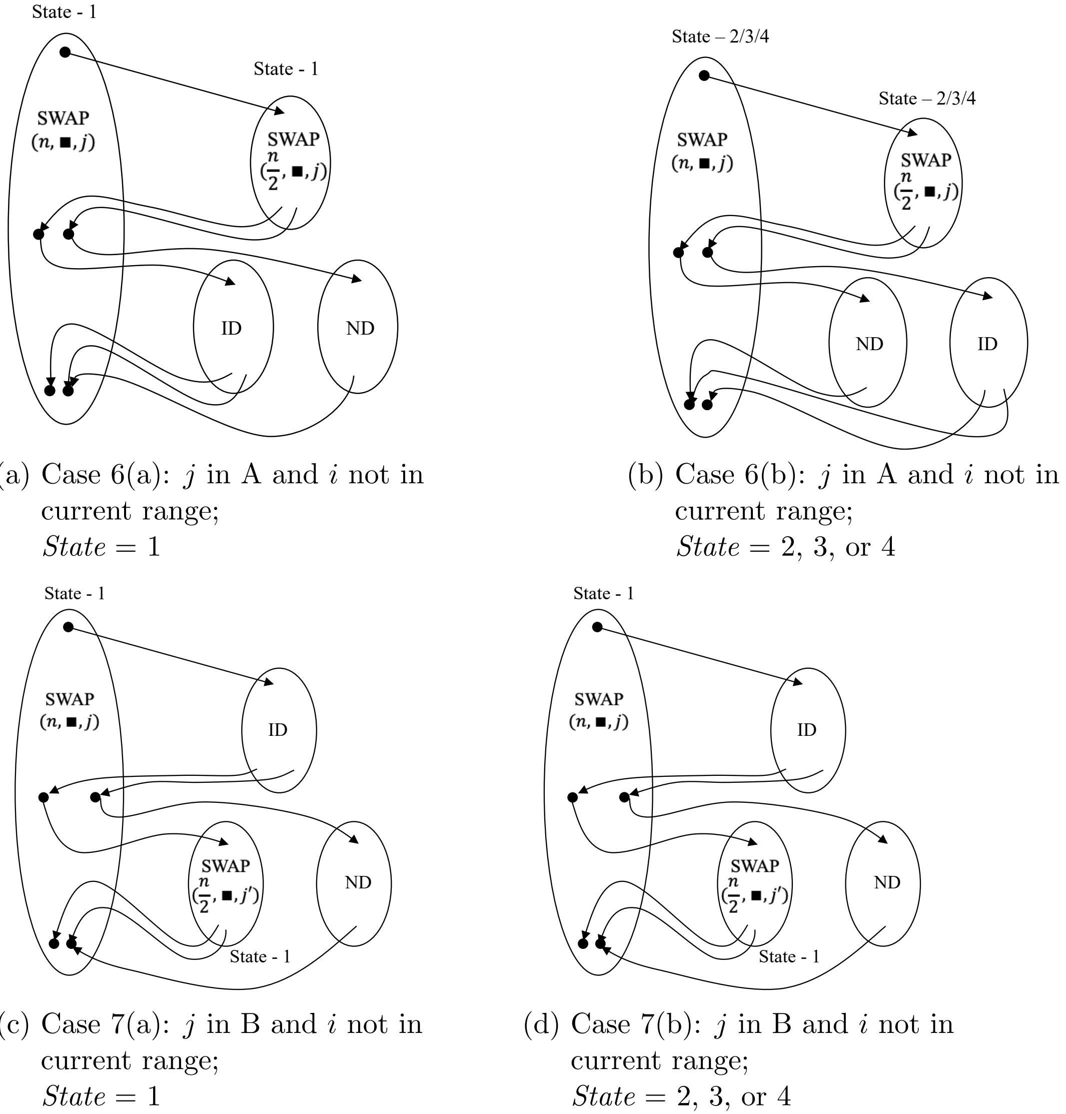


(c) Case 5(a): i in B and j not in current range; control-bit $= n$-1



(d) Case 5(b): i in B and j not in current range; control-bit $\neq n$-1

Figure J.4: The different cases of the $SWAP$ matrix construction, continued.



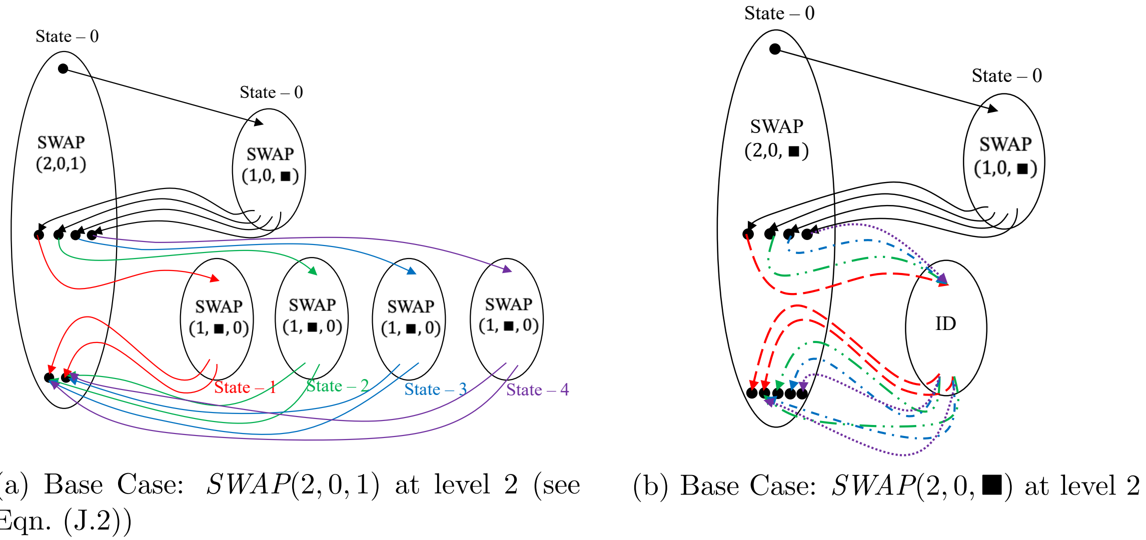
(a) Case 6(a): j in A and i not in current range; $State = 1$

(b) Case 6(b): j in A and i not in current range; $State = 2$, 3, or 4

(c) Case 7(a): j in B and i not in current range; $State = 1$

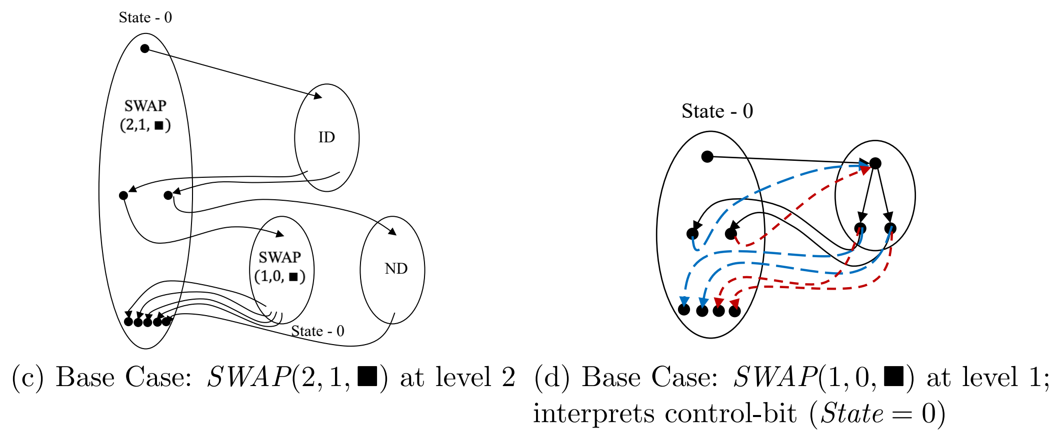
(d) Case 7(b): j in B and i not in current range; $State = 2$, 3, or 4

Figure J.5: The different cases of the $SWAP$ matrix construction, continued.



(a) Base Case: $SWAP(2, 0, 1)$ at level 2 (see Eqn. (J.2))

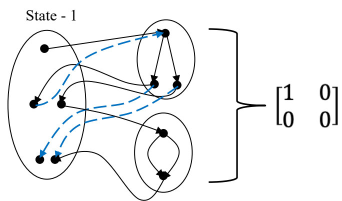
(b) Base Case: $SWAP(2, 0, \blacksquare)$ at level 2



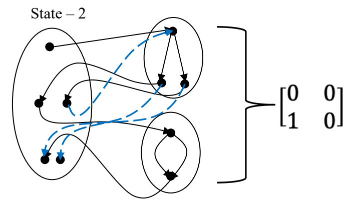
(c) Base Case: $SWAP(2, 1, \blacksquare)$ at level 2

(d) Base Case: $SWAP(1, 0, \blacksquare)$ at level 1; interprets control-bit ($State = 0$)

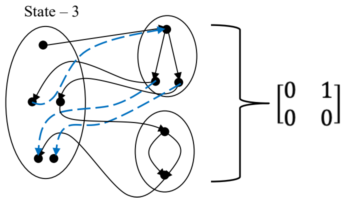
Figure J.6: Base cases for the construction of the $SWAP$ matrix. There are three base cases at level 2 with 2 bits (i.e., 4 Boolean variables) and five base cases at level 1 with 1 bit (i.e., 2 Boolean variables).



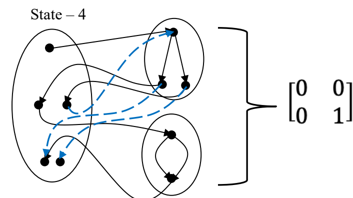
(a) Base Case: $SWAP(1, \blacksquare, 0)$ at level 1; interprets controlled-bit ($State = 1$)



(b) Base Case: $SWAP(1, \blacksquare, 0)$ at level 1; interprets controlled-bit ($State = 2$)



(c) Base Case: $SWAP(1, \blacksquare, 0)$ at level 1; interprets controlled-bit ($State = 3$)



(d) Base Case: $SWAP(1, \blacksquare, 0)$ at level 1; interprets controlled-bit ($State = 4$)

Figure J.7: Base cases for the construction of the $SWAP$ matrix, continued.

Appendix K: Time Complexity of Reduce

In this section, we give a bound on the time complexity of the call on Reduce (Alg. 14) in line [5] of BinaryApplyAndReduce (Alg. 11). Let C be the level-l proto-CFLOBDD on which Reduce is invoked, and C' be the level-l proto-CFLOBDD that is returned. The accounting is somewhat subtle because of three factors

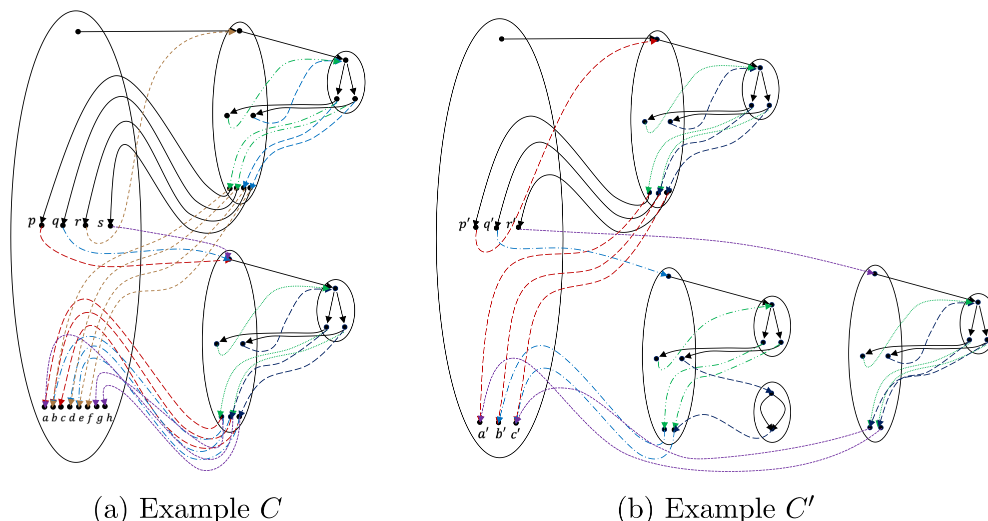
- hash-consing of groupings
- function caching of calls to Reduce and other functions
- for $C' = \mathit{Reduce}(C, \mathit{red})$ (where *red* is some reduction tuple), for their respective top-level groupings, g' and g, it is always the case that $|g'| \leq |g|$, yet $|C'|$ and $|C|$ have no fixed relationship: $|C'| < |C|$, $|C'| = |C|$ and $|C'| > |C|$ are all possible.[1]

The size measure $| \cdot |$ counts vertices and edges (and, for proto-CFLOBDDs, groupings—with no double-counting of shared groupings due to hash-consing).

In this section, we show that the time complexity of Reduce is bounded by $O(|C| \times |C'|)$, where when counting the time for operations, we consider the cost of function-caching operations (lookup and update) to be O(1).

We illustrate the point about there being no fixed relationship between C' and C with the following example, which shows that when Reduce is called on a proto-CFLOBDD, it can lead to both (i) less sharing of proto-CFLOBDDs in the resultant proto-CFLOBDD, and (ii) more sharing of proto-CFLOBDDs than in the input proto-CFLOBDD.

[1] We refer to the property that $|g'| \leq |g|$ as the *local-reduction property*, in contradistinction to the absence of a *global-reduction property* for $|C'|$ and $|C|$.



(a) Example C (b) Example C'

Figure K.1: $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$. The colors of the edges to proto-CFLOBDDs in C' correspond to the edges to the originating proto-CFLOBDDs in C.

Example K.1. Consider the level-2 proto-CFLOBDD C shown in Fig. K.1a, which has four middle vertices (p, q, r, s) and eight exit vertices (a, b, c, d, e, f, g, h). The A-connection of C (C_A) is a proto-CFLOBDD at level 1. C_A partitions the strings $\{0,1\}^2$ into $P1 = [\{00\}, \{01\}, \{10\}, \{11\}]$, i.e., C_A has four exit vertices and thus C has four middle vertices and four B-connections (C_{B1}, C_{B2}, C_{B3}, C_{B4}). C_{B1} partitions the strings $\{0,1\}^2$ into $P2 = [\{00\}, \{01, 10\}, \{11\}]$ and its exit vertices are connected to the exit vertices of C in the order (a, b, c). C_{B2} and C_{B4} both equal C_{B1}, but for C_{B2} and C_{B4} the three exit vertices are connected to the exit vertices of C in the orders (b, d, e), and (g, a, h), respectively. Finally, C_{B3} equals C_A, but for C_{B3} the four exit vertices are connected to C's exit vertices in the order (b, d, e, f). Consequently, C partitions the strings $\{0,1\}^4$ into $[\{0000, 1101, 1110\}, \{0001, 0010, 0100, 1000\}, \{0011\}, \{0101, 0110, 1001\}, \{0111, 1010\}, \{1011\}, \{1100\}, \{1111\}]$.

Let $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$, i.e., the vertices c, d, e, f, g, h are all mapped to exit vertex c. C' is shown in Fig. K.1b. C' has only three exit vertices (a', b', c'). Consider how C is "re-

duced” to C', which partitions the strings $\{0,1\}^4$ into [{0000,1101,1110}, {0001,0010,0100,1000},{0011,0101,0110,1001,0111,1010,1011,1100,1111}].

- C_{B1}'s exit vertices are mapped to (a, b, c), which leads to the call Reduce(C_{B1}, $[1, 2, 3]$) and thus C_{B1} does not change; that is, the first B-connection of C', C'_{B1}, is equal to C_{B1}. Its exit vertices are connected to the exit vertices (a', b', c') of C'. (As we will see below, C'_{B1} is also equal to C'_A, the A-connection of C'.)

- C_{B2}'s exit vertices are mapped to (b, c, c), which leads to the call Reduce(C_{B2}, $[1, 2, 2]$). Therefore, the second and third exit vertices are folded together, and this collapse affects the structure of the level-0 groupings as well, thereby creating a new proto-CFLOBDD, C'_{B2}, which partitions the strings $\{0,1\}^2$ into $[\{00\}, \{01, 10, 11\}]$. The exit vertices of C'_{B2} are mapped to exit vertices (b', c') of C'.

- C_{B3}'s exit vertices are mapped to (b, c, c, c), which leads to the call Reduce(C_{B3}, $[1, 2, 2, 2]$). Thus, the exit vertices of C_{B3} are collapsed to only two exit vertices, and the resulting proto-CFLOBDD partitions the strings $\{0,1\}^2$ into $[\{00\}, \{01, 10, 11\}]$, which are mapped to the exit vertices (b', c'). This result is identical to the result from Reduce(C_{B2}, $[1, 2, 2]$), and thus C' has only one copy of C'_{B2} with its exit vertices mapped to exit vertices (b', c') of C'.

- C_{B4}'s exit vertices are mapped to (c, a, c), which leads to the call Reduce(C_{B4}, $[1, 2, 1]$)—folding together the first and third exit vertices. This call creates yet another new proto-CFLOBDD, C'_{B3}, which partitions the strings $\{0,1\}^2$ into $[\{00, 11\}, \{01, 10\}]$. The exit vertices of C'_{B3} are mapped to exit vertices (c', a') of C'.

- Because the calls Reduce(C_{B2}, $[1, 2, 2]$) and Reduce(C_{B3}, $[1, 2, 2, 2]$) produce the same proto-CFLOBDD with the same return edges in C'—and because the calls

on Reduce arose in the B-connection of the same grouping in C—middle vertices (q, r) of C are folded together. This collapsing is propagated to the A-connection of C by the call Reduce$(C_A, [1, 2, 2, 3])$. The resulting proto-CFLOBDD has three exit vertices that partition the strings $\{0, 1\}^2$ into $[\{00\}, \{01, 10\}, \{11\}]$. This proto-CFLOBDD is identical to C'_{B1}—although their exit vertices are mapped to different vertices of C': the exit vertices of C'_{B1} are connected to exit vertices (a', b', c') of C', whereas the exit vertices of C'_A are connected to middle vertices (p', q', r') of C'.

We see from this example that a call $C' = \text{Reduce}(C, \mathit{red})$ can cause entirely new proto-CFLOBDDs to be created in C'; proto-CFLOBDDs that occur in C to occur in entirely different places in C'; proto-CFLOBDDs that occur in C to not occur in C'; and two or more proto-CFLOBDDs with identical sets of return edges to be combined into just a single occurrence when they arise in the B-connection of the same enclosing grouping. This example highlights the challenges for establishing a bound on the time complexity of Reduce—namely, both expansion and compaction of proto-CFLOBDDs can occur.

Because of the effects illustrated in Ex. K.1, the cost-bound argument we give is slightly indirect. At a high-level, it is structured as follows: we establish a relationship between Reduce(C, red) and that of a certain call on PairProduct (Thm. K.1). This approach is beneficial because we already know a time bound on PairProduct in terms of the product of the sizes of PairProduct's arguments (which is expressed more precisely in footnote 11). Thm. K.3 uses that bound to give an asymptotic bound on the time to perform Reduce(C, red) in terms of the product of the sizes of its input and output CFLOBDDs.

Theorem K.1. Let C and C' be two proto-CFLOBDDs such that $C' = \mathit{Reduce}(C, \mathit{red})$ for some reduction tuple *red*. Then $C = \text{PairProduct}(C, C')$.[2]

[2]To reduce clutter, we ignore the tuple of pairs of exit vertices that is returned by PairProduct (Alg. 12), except for two places in Thm. K.3.

Proof: We know that each proto-CFLOBDD at level k with m exit vertices partitions the space of strings $\{0,1\}^{2^k}$ into m groups (see §3.5). We make use of the properties of Reduce and PairProduct with respect to such partitions:

1. For every proto-CFLOBDD X and reduction tuple *red*, Reduce(X, *red*) produces a coarser partition of the exit languages of X defined by the mapping of *red* to X's exit vertices.
2. For every pair of proto-CFLOBDDs X and Y, $\mathit{PairProduct}(X, Y)$ produces the coarsest partition that refines both of the partitions corresponding to X and Y.

In particular, we consider the two-statement sequence

$$C' = \text{Reduce}(C, \mathit{red}); \tag{K.1}$$

$$\tilde{C} = \text{PairProduct}(C, C'); \tag{K.2}$$

C' created in Eqn. (K.1) represents a coarser partition of the strings in $\{0,1\}^n$ than C's partition. Because C' represents a coarser partition than C, the proto-CFLOBDD $\tilde{C}$ created in Eqn. (K.1) represents the same partition as C, and thus $\tilde{C}$ and C are equal by canonicity. □

In essence, Thm. K.1 shows that PairProduct(C, C') "undoes" all of the actions taken during Reduce(C, red).

Example K.2. Consider the result $\tilde{C}$ = PairProduct$(C,\ C')$ for C and C' from Ex. K.1. PairProduct$(C,\ C')$ is first called on the A-connections of the respective outermost groupings, followed by calls on B-connections.

- PairProduct$(C_A,\ C'_A)$ produces a proto-CFLOBDD whose exit vertices represent the coarsest partition of $\{0,1\}^2$ that refines both of the partitions corresponding to the exit vertices of C_A and C'_A (i.e., $[\{00\},\{01\},\{10\},\{11\}]$ and $[\{00\},\{01,10\},\{11\}]$, respectively). Hence, the new proto-CFLOBDD

$\tilde{C}_A$ is constructed such that the exit vertices of $\tilde{C}_A$ represent the partition $[\{00\}, \{01\}, \{10\}, \{11\}]$. PairProduct also returns a tuple of index-pairs indicating the B-connections on which PairProduct needs to be called. In this case, the returned tuple is $[[1, 1], [2, 2], [3, 2], [4, 3]]$. Mapping this result to the middle vertices of C and C', we obtain $[[p, p'], [q, q'], [r, q'], [s, r']]$. These pairs are processed left-to-right, generating calls to PairProduct on B-connections.

- PairProduct(C_{B1}, C'_{B1}) (corresponding to the pair $[p, p']$) creates proto-CFLOBDD $\tilde{C}_{B1}$ with three exit vertices corresponding to the partition $[\{00\}, \{01, 10\}, \{11\}]$, returning the tuple $[[1, 1], [2, 2], [3, 3]]$. Mapping this result to the exit vertices of C and C', the initial (as-yet incomplete) sequence of exit vertices of $\tilde{C}$ would be $[[a, a'], [b, b'], [c, c']]$.

- PairProduct(C_{B2}, C'_{B2}) (corresponding to the pair $[q, q']$) creates proto-CFLOBDD $\tilde{C}_{B2}$ with three exit vertices corresponding to the partition $[\{00\}, \{01, 10\}, \{11\}]$ (the same as $\tilde{C}_{B1}$), returning the tuple $[[1, 1], [2, 2], [3, 2]]$. Mapping this result to the exit vertices of C and C', the exit vertices of $\tilde{C}$ would be extended to be $[[a, a'], [b, b'], [c, c'], [d, c'], [e, c']]$, and the exit vertices of $\tilde{C}_{B2}$ would be connected to $[b, b']$, $[d, c']$, and $[e, c']$.

- PairProduct(C_{B3}, C'_{B2}) (corresponding to the pair $[r, q']$) creates proto-CFLOBDD $\tilde{C}_{B3}$ with four exit vertices corresponding to the partition $[\{00\}, \{01\}, \{10\}, \{11\}]$ (the same as $\tilde{C}_A$), returning the tuple $[[1, 1], [2, 2], [3, 2], [4, 2]]$. Mapping this result to the exit vertices of C and C', the exit vertices of $\tilde{C}$ would be extended to be $[[a, a'], [b, b'], [c, c'], [d, c'], [e, c'], [f, c']]$, and the exit vertices of $\tilde{C}_{B3}$ would be connected to $[b, b']$, $[d, c']$, $[e, c']$, and $[f, c']$.

- PairProduct(C_{B4}, C'_{B3}) (corresponding to the pair $[s, r']$) creates proto-CFLOBDD $\tilde{C}_{B4}$ with three exit vertices corresponding to the partition $[\{00\}, \{01, 10\}, \{11\}]$ (again, the same as $\tilde{C}_{B1}$), returning the tuple $[[1, 1], [2, 2], [3, 1]]$. Mapping this result to the exit vertices of C

and C', the final sequence of exit vertices of $\tilde{C}$ would be set to $[[a,a'],[b,b'],[c,c'],[d,c'],[e,c'],[f,c'],[g,c'],[h,c']]$, and the exit vertices of $\tilde{C}_{B4}$ would be connected to $[g,c']$, $[a,a']$, and $[h,c']$.

$\tilde{C}$ has eight exit vertices, four middle vertices, and each of the A-connections and B-connections of $\tilde{C}$ and C are connected to isomorphic proto-CFLOBDDs. Consequently, $\tilde{C} = C$ up to isomorphism. Because hash-consing enforces that the members of each isomorphism class have a unique representation in memory, PairProduct(C, C') would return a pointer to C.

Lemma K.2. (Local-Reduction Property). Let C and C' be two proto-CFLOBDDs such that $C' = \mathit{Reduce}(C, \mathit{red})$ for some reduction tuple *red*, and let g and g' be their respective outermost groupings. Then $|g'| \leq |g|$.

Proof: The size of a grouping is equal to the number of entry, middle, and exit vertices, plus the number of A-connection and B-connection edges and return edges. Because g' is obtained by reducing g with respect to *red*, the number of exit vertices in g' can be no more than the number in g. Moreover, *Reduce* can never cause there to be more B-connections in g' than in g, but it can cause some B-connections of g to be folded together in g'; thus, the number of middle vertices in g' can be no more than the number in g. Similarly, for the A-connection of g' and all the B-connections of g', the number of return edges can be no more than the number of return edges in the corresponding A-/B-connections in g. Consequently, $|g'| \leq |g|$. $\square$

Example K.3. Consider the proto-CFLOBDDs C and C' from Fig. K.1. The size of the level-2 grouping g equals 1 (entry-vertex) + 4 (middle vertices) + (1 + 3) (1st B-connection) + (1 + 3)(2nd B-connection) + (1 + 4) (3rd B-connection) + (1 + 3)(4th B-connection) + 8 (exit vertices) = 30.

The size of g' equals 1 (entry-vertex) + 3 (middle vertices) + (1 + 3) (1st B-connection) + (1 + 2)(2nd B-connection) + (1 + 2) (3rd B-connection) + 3 (exit vertices) = 17.

Thus, $|g'| \leq |g|$, whereas $68 = |C'| > |C| = 66$.

We now turn to the question of bounding the time complexity of Reduce. Whereas Thm. K.1 showed that PairProduct(C, C') "undoes" all of the actions taken during Reduce(C, red), Thm. K.3 shows that for every action in Reduce(C, red), there is an action of at least the same cost in PairProduct(C, C'). Consequently, the time to perform Reduce(C) is bounded by the time that it would take to perform PairProduct(C, C'), which is $O(|C| \times |C'|)$.

Theorem K.3. Let C and C' be two proto-CFLOBDDs such that $C' =$ *Reduce*(C, red) for some reduction tuple *red*. Let *Cost*(Reduce(C)) and *Cost*$(PP(C, C'))$ denote the costs of Reduce(C, red) and *PairProduct*(C, C'), respectively. Then *Cost*(*Reduce*(C)) $\leq$ *Cost*$(PP(C, C'))$.

Proof: The proof is by induction on the level k of proto-CFLOBDDs C and C'.

Base case: $(k = 0)$ Consider the following table,

C	*red*	$C' =$ Reduce(C, red)	PairProduct(C, C')
ForkGouping	$[1, 1]$	DontCareGrouping	[ForkGrouping, $([1, 1], [2, 1])$]
DontCareGrouping	$[1]$	DontCareGrouping	[DontCareGrouping, $([1, 1])$]
ForkGouping	$[1, 2]$	ForkGouping	[ForkGouping, $([1, 1], [2, 2])$]
DontCareGrouping	$--$	ForkGouping	Not Applicable

The last line in the table cannot arise because there is no reduction tuple that can be used to reduce a DontCareGrouping to a Fork Grouping. In each of the other three cases in the table, PairProduct(C, C') returns a tuple that has C as the first component.

Moreover, the results produced by Reduce(C) and PairProduct(C, C') are of constant size, and could be implemented by table lookup. The return value from PairProduct(C, C') is larger than the return value from Reduce(C, red), which justifies saying that *Cost*(*Reduce*(C)) $\leq$ *Cost*$(PP(C, C'))$.

Induction step:

Induction Hypothesis: Assume that for all level-k proto-CFLOBDDs C'_k and C_k for which $C'_k = \mathit{Reduce}(C_k, \mathit{red})$, for some reduction tuple *red*, $\mathit{Cost}(\mathit{Reduce}(C_k)) \leq \mathit{Cost}(PP(C_k, C'_k))$.

Consider two level-k+1 proto-CFLOBDDs, C'_{k+1} and C_{k+1}, such that $C'_{k+1} = \text{Reduce}(C_{k+1}, \mathit{red})$. The proof breaks down into the following three cases:

(i) A-connections. PairProduct is first called recursively on $C_{k+1}.A$ and $C'_{k+1}.A$—i.e., the level-k A-connections of C_{k+1} and C'_{k+1}, respectively. By the construction of C'_{k+1} from C_{k+1}, we know that $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, \mathit{red}_A)$ for some reduction tuple red_A. Thus, by the induction hypothesis,

$$\mathit{Cost}(\mathit{Reduce}(C_{k+1}.A)) \leq \mathit{Cost}(PP(C_{k+1}.A, C'_{k+1}.A)). \tag{K.3}$$

(ii) B-connections. The return value from the call on PairProduct$(C_{k+1}.A, C'_{k+1}.A)$ considered in the previous case is actually a tuple $[\tilde{C}_{k+1}.A, \mathit{midVertexPairs}]$. By the construction of C'_{k+1} from C_{k+1}, we know that $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, \mathit{red}_A)$ for some reduction tuple red_A, and thus by Thm. K.1, $\tilde{C}_{k+1}.A = C_{k+1}.A$.

For every $(i, j) \in \mathit{midVertexPairs}$, PairProduct is called recursively on $C_{k+1}.B[i]$ and $C'_{k+1}.B[j]$, which are level-k proto-CFLOBDDs. To be able to invoke the induction hypothesis, we must establish that $C'_{k+1}.B[j] = \mathit{Reduce}(C_{k+1}.B[i], \mathit{red}_{B[i]})$, for some $\mathit{red}_{B[i]}$.

We now consider the meaning of the pairs $(i, j) \in \mathit{midVertexPairs}$ from the standpoint of the language partitions used in Thm. K.1. Because C_{k+1} represents a finer partition of the strings in $\{0,1\}^{2^{k+1}}$ than C'_{k+1}, the ith exit vertex of $C_{k+1}.A$ represents a finer partition of the strings in $\{0,1\}^{2^k}$ than the jth exit vertex of $C'_{k+1}.A$. Thus, in general, there can be multiple exit vertices $i_1, i_2, \ldots, i_p$ of $C_{k+1}.A$ whose language partitions were combined to create the language partition of the jth exit vertex of $C'_{k+1}.A$.

Because these vertices are exit vertices of A-connections, we can equivalently refer to the set $\{i_1, i_2, \ldots, i_p\}$ of middle vertices of C_{k+1} and the jth middle vertex of C'_{k+1}. The reason this combining of languages took place during Reduce(C_{k+1}, *red*) can only be because there were calls on Reduce($C_{k+1}.B[i_1]$, red_1), Reduce($C_{k+1}.B[i_2]$, red_2), ..., Reduce($C_{k+1}.B[i_p]$, red_p), for which the results were all equal to $C'_{k+1}.B[j]$. (The fifth bullet point of Ex. K.1 illustrates how calls to Reduce on two different B-connections in the same grouping yield the same result, which folds together two middle vertices of the grouping—thereby unioning their language partitions in the proto-CFLOBDD returned by Reduce.) Consequently, by the induction hypothesis,

$$\begin{aligned}
&Cost(Reduce(C_{k+1}.B[i_1])) \leq Cost(PP(C_{k+1}.B[i_1], C'_{k+1}.B[j])) \\
&Cost(Reduce(C_{k+1}.B[i_2])) \leq Cost(PP(C_{k+1}.B[i_2], C'_{k+1}.B[j])) \\
&\qquad\qquad\qquad\qquad \ldots \\
&Cost(Reduce(C_{k+1}.B[i_p])) \leq Cost(PP(C_{k+1}.B[i_p], C'_{k+1}.B[j]))
\end{aligned} \tag{K.4}$$

Let e_A denote the number of exit vertices of $C_{k+1}.A$ (which is also the number of middle vertices of C_{k+1}). These inequalities can be expressed more succinctly by observing that for each index i, $1 \leq i \leq e_A$ on the left-hand side (corresponding to an A-connection language-partition of $C_{k+1}.A$), there is a unique j to use on the right-hand side of the inequality. (Index j corresponds to the coarsened A-connection language-partition of $C'_{k+1}.A$.) Let *reductum* denote this index map: i.e., $j = \text{reductum}(i)$. We can now rewrite Eqn. (K.4) as

$$Cost(Reduce(C_{k+1}.B[i])) \leq Cost(PP(C_{k+1}.B[i], C'_{k+1}.B[\text{reductum}(i)])) \tag{K.5}$$

(iii) Overall cost. Let g' and g denote the outermost groupings (at level $k+1$) of C'_{k+1} and C_{k+1}, respectively. Reduce and PairProduct each make a call on `RepresentativeGrouping` at the end of their computations to hash-cons the outermost grouping that has been constructed. The time complexity of a call on

`RepresentativeGrouping` is dominated by the cost of computing the grouping's hash value, and thus the costs in Reduce and PairProduct are linear in $|g'|$ and $|g|$, respectively. By Lem. K.2, we know that $|g'| \leq |g|$, and thus the cost of the call on `RepresentativeGrouping` in Reduce is no more than the cost of the call in PairProduct.

Finally, using Lem. K.2 and Eqns. (K.3) and (K.5), we obtain the desired result:

$$
\begin{aligned}
\mathit{Cost}(\mathit{Reduce}(C_{k+1})) &= |g'| + \sum_{i=1}^{e_A} \mathit{Cost}(\mathit{Reduce}(C_{k+1}.B[i])) + \mathit{Cost}(\mathit{Reduce}(C_{k+1}.A)) \\
&\leq |g| + \sum_{i=1}^{e_A} \mathit{Cost}(\mathit{Reduce}(C_{k+1}.B[i])) + \mathit{Cost}(\mathit{Reduce}(C_{k+1}.A)) \\
&= |g| + \sum_{i=1}^{e_A} \mathit{Cost}(PP(C_{k+1}.B[i], C'_{k+1}.B[\text{reductum}(i)])) \\
&+ \mathit{Cost}(PP(C_{k+1}.A, C'_{k+1}.A)) \\
&= \mathit{Cost}(PP(C_{k+1}, C'_{k+1})).
\end{aligned}
$$

□

Appendix L: Constructing a Canonical WCFLOBDD from a Decision Tree

The construction of a WCFLOBDD for a function f from the decision-tree representation of f is a two-step process: (i) constructing a weighted decision tree (WDT) from the decision tree for f, and (ii) constructing a WCFLOBDD from the WDT for f.

The construction of a WDT from a decision tree has the same flavor as constructing a WBDD, as discussed in Vrudhula et al. (1996) and Zulehner and Wille (2020).

Construction 2. [Decision Tree to WDT]

Given a decision tree T that represents a function f, perform the following actions recursively, traversing T in post-order (i.e., performing actions just before returning), thereby creating the WDT bottom-up.

1. *If the current node n is a leaf node of T with terminal value v, create a new leaf node n' with terminal value v', where*

$$v' = \begin{cases} \bar{1} & \textit{if } v \neq \bar{0} \\ \bar{0} & \textit{otherwise} \end{cases}$$

   Return the tuple $\langle v, n' \rangle$.

2. *If n is an internal node of T, and tuples $\langle v_1, n_1 \rangle$ and $\langle v_2, n_2 \rangle$ are the tuples returned from the recursive calls to the left child and right child, respectively, create a new internal node n_0 with (i) left child n_1 and right child n_2, and (ii) edge weights (lw, rw) as follows:*

$$(lw, rw) = \begin{cases} (\bar{1}, v_1^{-1} \cdot v_2) & \textit{when } v_1 \neq \bar{0} \\ (\bar{0}, \bar{1}) & \textit{otherwise} \end{cases}$$

Let v_0 be defined as follows:

$$v_0 = \begin{cases} v_1 & \textit{when } v_1 \neq \bar{0} \\ v_2 & \textit{otherwise} \end{cases}$$

Return the tuple $\langle v_0, n_0 \rangle$.

The result is a WDT $\langle v', n' \rangle$, where v' is the factor weight. □

Construction 3. [WDT to WCFLOBDD]

Definition L.1. A *proto-Weighted Decision Tree* (proto-WDT) is a WDT whose "leaves" are either (i) all terminal values $\bar{0}$ or $\bar{1}$, or (ii) all proto-WDTs that are of the same height and obey the edge-weight conditions of a WDT.

Just as inductive arguments about WCFLOBDDs have to be couched in terms of proto-WCFLOBDDs, the main part of the construction below focuses on proto-WDTs of a given WDT (T), and shows how to construct a proto-WCFLOBDD from a proto-WDT. That construction provides a way to construct a WCFLOBDD from a WDT as a special case.

One property of WDTs that is different for proto-WDTs is the number of equivalence classes of their leaves. Suppose that WDT T has 2^{2^n} leaves. T has at most two kinds of leaves, $\bar{0}$ and $\bar{1}$, and thus just one or two equivalence classes of leaves. In contrast, suppose that T' is a proto-WDT of T of height 2^k and 2^{2^k} leaves. T' can have between 1 and 2^{2^k} equivalence classes of leaves.

For convenience, in the discussion below, when we refer to a proto-WCFLOBDD w, we mean a proto-WCFLOBDD proper, together with a value tuple v where $|v|$ is equal to the number of exit vertices of the head-grouping of w.

The following recursive procedure describes how to convert a proto-WDT T' of height 2^k and 2^{2^k} leaves into a level-k proto-WCFLOBDD whose value tuple consists of an enumeration of the leaf equivalence classes e' of T'. The enumeration of the $|e'|$

equivalence-class representatives respects the relative ordering of their first occurrences in a left-to-right sweep over the leaves of T'.

1. *Base case (when $k = 0$): There are two cases, depending on whether the height-1 proto-WDT has one leaf equivalence class, $\{v\}$, or two, $\{v_1\}$ and $\{v_2\}$. In the first case, create a* `DontCareGrouping` *(a level-0 proto-WCFLOBDD), and attach $[v]$ as the value tuple. In the second case, create a* `ForkGrouping` *(a level-0 proto-WCFLOBDD), and then attach $[v_1, v_2]$ as the value tuple. In both cases, the left-edge and right-edge weights of the level-0 grouping are copied from the weights of the left and right edges of the proto-WDT node.*

2. *For each of the $2^{2^{k-1}}$ proto-WDTs of height 2^{k-1} in the lower half of T', construct—via a recursive application of the construction—$2^{2^{k-1}}$ level-$(k-1)$ proto-WCFLOBDDs. The leaf values of the height-2^{k-1} proto-WDTs are carried over from the leaf values of T'.*

   These proto-WCFLOBDDs are then partitioned into some number $e^{\#} \geq 1$ of equivalence classes of equal proto-WCFLOBDDs. A representative of each class is retained, and the others discarded. Each of the $2^{2^{k-1}}$ "leaves" of the upper half of proto-WDT T' is labeled with the appropriate equivalence-class representative (for the subtree of the lower half of T' that begins there). These proto-WCFLOBDD-valued leaves serve as the leaf values of the upper half of proto-WDT T' when the construction process is applied recursively to the upper half in step 3.

   The enumeration $1 \ldots |e^{\#}|$ of the equivalence-class representatives respects the relative ordering of their first occurrences in a left-to-right sweep over the leaves of the upper half of T'.

3. *Construct—via a recursive application of the procedure—a level-$(k$–$1)$ proto-WCFLOBDD A' for the proto-WDT consisting of the upper half of T' (with the WCFLOBDDs constructed in step 2 as the leaf values).*

4. *Construct a level-k proto-WCFLOBDD from the level-$(k–1)$ proto-WCFLOBDDs created in steps 2 and 3. The level-k grouping is constructed as follows:*

   (a) *The A-connection points to the level-$(k–1)$ proto-WCFLOBDD of proto-WCFLOBDD A' constructed in step 3.*

   (b) *The $|e^{\#}|$ middle vertices correspond to the equivalence classes formed in step 2 (in the enumeration order $1 \ldots |e^{\#}|$ of step 2).*

   (c) *The A-connection return tuple is the identity map back to the middle vertices (i.e., the tuple $[1..|e^{\#}|]$).*

   (d) *The B-connections point to the level-$(k–1)$ proto-WCFLOBDDs of the $|e^{\#}|$ equivalence-class representatives constructed in step 2, in the enumeration order $1 \ldots |e^{\#}|$.*

   (e) *The exit vertices of the proto-WCFLOBDD correspond to the equivalence classes e' of the leaves of T', in the enumeration order $1 \ldots |e'|$.*

   (f) *The B-connection return tuples connect (i) the exit vertices of the level-$(k-1)$ groupings of B-connections to (ii) the level-k grouping's exit vertices that were created in step 4e. The connections are made according to matching leaf equivalence classes from T'.*

   (g) *Consult a table of all previously constructed level-k groupings to determine whether the grouping constructed by steps 4a–4f duplicates a previously constructed grouping. If so, discard the present grouping and switch to the previously constructed one; if not, enter the present grouping into the table.*

5. *To create a proto-WCFLOBDD with value tuples from the proto-WCFLOBDD (without value tuples) constructed in step 4, we use a value tuple consisting of the leaf equivalence classes e' of T', listed in enumeration order (i.e., in the ordering of first occurrences in a left-to-right sweep over the leaves of T').*

To construct a WCFLOBDD for a WDT T, we merely consider T to be a proto-WDT with each leaf labeled by $\bar{0}$ or $\bar{1}$. Note that the leaf equivalence classes will be one of $[\bar{0}]$, $[\bar{1}]$, $[\bar{0}, \bar{1}]$, or $[\bar{1}, \bar{0}]$ (which becomes the value tuple of the constructed WCFLOBDD). $\square$

Appendix M: Algorithms for WCFLOBDDs

M.1 Pointwise Multiplication

This section gives two of the algorithms used in pointwise multiplication for WCFLOBDDs.

- Alg. 63 reduces a grouping based on return tuples or the value tuple, and as a by-product ensures canonicity of the resulting WCFLOBDD.
- Alg. 65 determines the position for a B-connection in a grouping being constructed, reusing one if it is already present in the grouping.

M.2 Pointwise Addition

The pseudo-code for pointwise addition of two WCFLOBDDs is given in Algs. 66, 67, and 68.

M.3 Kronecker Product

The pseudo-code for Kronecker Product on WCFLOBDDs is given as Algs. 69 and 70. (In Alg. 69, $\bowtie$ denotes the operation to interleave two variable orderings.)

M.4 Matrix Multiplication

Pseudo-code for the matrix-multiplication algorithm for WCFLOBDDs is shown in Algs. 71, 72, 73, and 74.

M.5 Sampling

A WCFLOBDD with no non-negative edge weights can be considered to represent a discrete distribution over the set of assignments to the Boolean variables. An

Algorithm 63: (WCFLOBDDs) Reduce

```
Input: Grouping g, ReductionTuple reductionTuple, ValueTuple
       valueTuple
Output: Grouping g' that is "reduced," weight w (factor weight for g')
1  begin
       // Test whether any reduction actually needs to be carried
          out
2      if reductionTuple == [1..|reductionTuple|] and each element in
        valueTuple equals v ≠ 0̄ then
3          return [g, v];
4      end
5      if every element in valueTuple == 0̄ then return
        [ConstantZeroProtoCFLOBDD(g.level), 0̄];
6      if g is fork grouping, reductionTuple = [1,2] and let valueTuple =
        [v_1, v_2] then
7          if v_1 == 0̄ then
8              ForkGrouping g' = new ForkGrouping(0̄, 1̄); return [g', v_2];
9          else
10             ForkGrouping g' = new ForkGrouping(1̄, v_1^{-1} v_2); return [g',
                v_1];
11         end
12     end
13     if g is fork grouping, reductionTuple = [1,1] and let valueTuple =
        [v_1, v_2] then
14         if v_1 == 0̄ then
15             DontCareGrouping g' = new DontCareGrouping(0̄, 1̄); return
                [g', v_2];
16         else
17             DontCareGrouping g' = new DontCareGrouping(1̄, v_1^{-1} v_2);
                return [g', v_1];
18         end
19     end
20     InternalGrouping g' = new InternalGrouping(g.level);
21     g'.numberOfExits = |{x : x ∈ reductionTuple}|;
```

assignment—or equivalently, the corresponding matched path—is considered to be an elementary event. The probability of a matched path p is the weight of p divided by the sum of the weights of all matched paths of the WCFLOBDD.

Algorithm 64: (WCFLOBDDs) Reduce (continued from Alg. 63)

```
22
23     Tuple reductionTupleA = []; Tuple valueTupleA = [];
24     for i ← 1 to g.numberOfBConnections do
25         Tuple deducedReturnClasses = [reductionTuple(v) : v ∈
             g.BReturnTuples[i]];
26         Tuple×Tuple [inducedReturnTuple, inducedReductionTuple] =
             CollapseClassesLeftmost(deducedReturnClasses);
27         Tuple inducedValueTuple = [deducedReturnClasses(i) : i ∈
             [1..|deducedReturnClasses|]];
28         Grouping×Weight [h, w_B] = Reduce(g.BConnection[i],
             inducedReductionTuple, inducedValueTuple);
29         int position = InsertBConnection(g', h, inducedReturnTuple);
30         reductionTupleA = reductionTupleA || position;
31         valueTupleA = valueTupleA || w_B;
32     end
33     Tuple×Tuple [inducedReturnTuple, inducedReductionTuple] =
         CollapseClassesLeftmost(reductionTupleA);
34     Tuple inducedValueTuple = [reductionTupleA(i) : i ∈
         [1..|reductionTupleA|]];
35     Grouping×Weight [h', w] = Reduce(g.AConnection,
         inducedReductionTuple, inducedValueTuple);
36     g'.AConnection = h';
37     g'.AReturnTuple = inducedReturnTuple;
38     return [RepresentativeGrouping(g'), w];
39 end
```

M.5.1 Weight Computation

To sample an assignment directly from the WCFLOBDD representation of the function, we first need to compute the weight corresponding to every exit vertex. The weight of an exit vertex e of a grouping g is the sum of the weights of all matched paths through the proto-WCFLOBDD headed by g that lead to e. This information can be computed recursively by (i) computing the weight of every middle vertex of g for all matched paths from the entry vertex to middle vertices, and then (ii) computing the weight of every exit vertex of g for all matched paths from the middle vertices to

Algorithm 65: (WCFLOBDDs) InsertBConnection

Input: InternalGrouping g, Grouping h, ReturnTuple returnTuple

Output: int – Insert (h, ReturnTuple) as the next B-connection of g, if they are a new combination; otherwise return the index of the existing occurrence of (h, ReturnTuple)

1 **begin**
2 **if** *there exists* $i \in [1..g.\text{numberOfBConnections}]$ *such that g.BConnections[i] == h && g.BReturnTuples[i] == returnTuple* **then return** i;
3 g.numberOfBConnections = g.numberOfBConnections + 1;
4 g.BConnections[g.numberOfBConnections] = h;
5 g.BReturnTuples[g.numberOfBConnections] = returnTuple;
6 **return** g.numberOfBConnections;
7 **end**

Algorithm 66: (WCFLOBDDs) Pointwise Addition

Input: WCFLOBDDs n1 = $\langle fw1, h1, vt1\rangle$, n2 = $\langle fw2, h2, vt2\rangle$

Output: CFLOBDD n = n1 + n2

1 **begin**
`// Perform ‘‘weighted’’ cross product`
2 Grouping×Tuple [g,pt] = WeightedPairProduct(*h1*,*h2*, *fw1*, *fw2*);
3 ValueTuple deducedValueTuple = [c1 · vt1[i1] + c2 · vt2[i2] : $[\langle (c1, i1), (c2, i2)\rangle] \in$ pt];
`// Collapse duplicate leaf values, folding to the left`
4 Tuple×Tuple [inducedReturnTuple,inducedReductionTuple] = CollapseClassesLeftmost(deducedValueTuple) ;
5 Tuple inducedValueTuple = one of $[\bar{1}, \bar{0}], [\bar{0}, \bar{1}], [\bar{1}], [\bar{0}]$ based on inducedReturnTuple ;
6 Grouping×Weight [g’, *fw*] = Reduce(g, inducedReductionTuple, deducedValueTuple) ;
7 WCFLOBDD n = RepresentativeCFLOBDD(*fw*, g’, inducedValueTuple) ;
8 **return** n;
9 **end**

the exit vertices, (iii) combining this information to obtain the weight of every exit vertex of g for matched paths from entry vertex to exit vertices of g.

Algorithm 67: (WCFLOBDDs) WeightedPairProduct

```
   Input: Groupings g1, g2; Weights p1, p2
   Output: Grouping g: g1 + g2; Tuple ptAns: tuple of pairs of exit
           vertices with corresponding weights
1  begin
2    if g1 is ConstantZeroCFLOBDD then return [ g2, [⟨(1̄, 1), (0̄, k)⟩ : k
       ∈ [1..g2.numberOfExits]] ;
3    if g2 is ConstantZeroCFLOBDD then return [ g1, [⟨(0̄, k), (1̄, 1)⟩ : k
       ∈ [1..g1.numberOfExits]] ;
4    if g1 == g2 then return [ g1, [⟨(1̄, k), (1̄, k)⟩ : k ∈
       [1..g2.numberOfExits]] ;
     // Similar for other base cases, with appropriate weights
        and exit vertices pairings
5    if g1 and g2 are fork groupings then
6      ForkGrouping g = new ForkGrouping(1,1);
7      return [ g, [⟨(p1 · g1.lw, 1), (p2 · g2.lw, 1)⟩,
         ⟨(p1 · g1.rw, 2), (p2 · g2.rw, 2)⟩] ];
8    end
     // Pair the A-connections
9    Grouping×Tuple [gA,ptA] = WeightedPairProduct(g1.AConnection,
       g2.AConnection, p1, p2);
10   InternalGrouping g = new InternalGrouping(g1.level);
11   g.AConnection = gA ;
12   g.AReturnTuple = [1..|ptA|]; // Represents the middle vertices
13   g.numberOfBConnections = |ptA| ;
```

Consider a grouping g at level l with e exit vertices. Suppose that $g.AConnection$ has p exit vertices; suppose that $g.BConnections[j]$ (where $1 \leq j \leq p$) has k_j exit vertices; and let $g.BReturnTuples[j]$ be the return edges from $g.BConnections[j]$'s exit vertices to g's exit vertices. To compute Step (i), we recursively call the weight-computation procedure for $g.AConnection$, which yields a vector of weights v_A of size $1 \times p$. For Step (ii), the vectors obtained from recursive calls on the weight-computation procedure for the p B-connections of g are used to create a matrix M_B of size $p \times e$, in which the jth row is the vector of weights from the jth middle vertex of g to g's exit vertices. (The details are given in the next paragraph.) For Step (iii), the vector-matrix product $v_A \times M_B$ yields g's weight vector, of size $1 \times e$.

Algorithm 68: (WCFLOBDDs) WeightedPairProduct (cont.)

```
14
       // Pair the B-connections, but only for pairs in ptA
       // Descriptor of pairings of exit vertices
15     Tuple ptAns = [];
       // Create a B-connection for each middle vertex
16     for j ← 1 to |ptA| do
17         Grouping×Tuple [gB,ptB] =
            WeightedPairProduct(g1.BConnections[ptA(j)(1)(2)],
            g2.BConnections[ptA(j)(2)(2)], ptA(j)(1)(1), ptA(j)(2)(1));
18         g.BConnections[j] = gB;
           // Now create g.BReturnTuples[j], and augment ptAns as
              necessary
19         g.BReturnTuples[j] = [];
20         for i ← 1 to |ptB| do
21             c1 = g1.BReturnTuples[ptA(j)(1)(2)](ptB(i)(1)(2));      // an
                exit vertex of g1
22             c2 = g2.BReturnTuples[ptA(j)(2)(2)](ptB(i)(2)(2));      // an
                exit vertex of g2
23             f1 = g1.BReturnTuples[ptA(j)(1)(2)](ptB(i)(1)(1));      // an
                associated weight of g1
24             f2 = g2.BReturnTuples[ptA(j)(2)(2)](ptB(i)(2)(1));      // an
                associated weight of g2
25             if ⟨(f1, c1), (f2, c2)⟩ ∈ ptAns then        // Not a new exit
                vertex of g
26                 index = the k such that ptAns(k) == ⟨(f1, c1), (f2, c2)⟩;
27                 g.BReturnTuples[j] = g.BReturnTuples[j] || index;
28             else                  // Identified a new exit vertex of g
29                 g.numberOfExits = g.numberOfExits + 1;
30                 g.BReturnTuples[j] = g.BReturnTuples[j] ||
                    g.numberOfExits;
31                 ptAns = ptAns || ⟨(f1, c1), (f2, c2)⟩;
32             end
33         end
34     end
35     return [RepresentativeGrouping(g), ptAns];
36 end
```

Algorithm 69: (WCFLOBDDs) Kronecker Product

Input: WCFLOBDDs n1 = $\langle fw1, g1, vt1\rangle$, n2 = $\langle fw2, g2, vt2\rangle$ with variable ordering of n1: $x \bowtie y$ and n2: $w \bowtie z$

Output: WCFLOBDD n = n1 $\otimes$ n2 with variable ordering of n: $(x||w) \bowtie (y||z)$

```
1  begin
2      if fw1 == 0̄ or fw2 == 0̄ then return
         ConstantZeroProtoCFLOBDD(n1.level);
3      e = index of 0̄ in vt1 (-1 if no such occurrence);
4      e' = index of 0̄ in vt2 (-1 if no such occurrence);
5      Grouping g = KroneckerProductOnGrouping(g1, g2, e, e');
6      ValueTuple vt;
7      if e == -1 then vt = vt2;
8      if e == 1 then vt = [0̄, 1̄];
9      if e == 2 and vt2 == [1̄] then
10         vt = vt2 || [0̄];
11     else
12         vt = vt2;
13     end
14     return RepresentativeCFLOBDD(fw1 · fw2, g, vt);
15 end
```

At level-0, if the grouping is a *ForkGrouping* with left-edge and right-edge weights (lw, rw), then the weight vector is $[lw, rw]$. If the grouping is *DontCareGrouping*, then the weight vector is $[lw + rw]$.

Because the exit vertices of $g.BConnections[j]$ are connected to g's exit vertices via $g.BReturnTuples[j]$, the jth row of M_B is the product of the weight vector for $g.BConnections[j]$ (of size $1 \times k_j$) and a "permutation matrix" $PM^{g.BReturnTuples[j]}$ (of size $k_j \times e$). Each entry of PM is either 0 or 1; each row must have exactly one 1; and each column must have at most one 1.

This definition can be stated equationally, where the expression in large brackets represents M_B.

Algorithm 70: (WCFLOBDDs) KroneckerProductOnGrouping

```
  Input: Groupings g1, g2; e (exit of g1 that leads to 0̄), e' (exit of g2 that
         leads to 0̄)
  Output: Grouping g' such that g' = g1 ⊗ g2
1 begin
2   InternalGrouping g' = new InternalGrouping(g.level + 1);
3   g'.AConnection = g1;
4   g'.AReturnTuple = [1..|g1.numberOfExits|];
5   g'.numberOfBConnections = |g1.numberOfExits|;
6   if e == -1 then
7     g'.BConnection[1] = g2;
8     g'.BReturnTuple[1] = [1..|g2.numberOfExits|];
9     g'.numberOfExits = |g2.numberOfExits|;
10  end
11  if e == 1 then
12    g'.BConnection[1] = ConstantZeroProtoCFLOBDD(g.level);
13    g'.BReturnTuple[1] = [1];
14    g'.BConnection[2] = g2;
15    if e' == -1 then g'.BReturnTuple[2] = [2] ;
16    if e' == 1 then g'.BReturnTuple[2] = [1,2] ;
17    else g'.BReturnTuple[2] = [2,1] ;
18    g'.numberOfExits = 2;
19  end
20  if e == 2 then
21    g'.BConnection[1] = g2;
22    g'.BReturnTuple[1] = [1..|g2.numberOfExits|];
23    g'.BConnection[2] = ConstantZeroProtoCFLOBDD(g.level);
24    if e' == -1 then g'.BReturnTuple[2] = [2] ;
25    if e' == 1 then g'.BReturnTuple[2] = [1] ;
26    else g'.BReturnTuple[2] = [2] ;
27    g'.numberOfExits = 2;
28  end
29  return RepresentativeGrouping(g');
30 end
```

$$weightOfExit^{g}_{1\times e} = \begin{cases} [lw, rw]_{1\times 2} & \text{if g} = \textit{Fork} \\ [lw + rw]_{1\times 1} & \text{if g} = \textit{DontCare} \\ W & \text{otherwise} \end{cases}$$

Algorithm 71: (WCFLOBDDs) Matrix Multiplication

```
    Input: WCFLOBDDs n1 = ⟨fw1, g1, vt1⟩, n2 = ⟨fw2, g2, vt2⟩
    Output: WCFLOBDD n = n1 × n2
 1  begin
 2      Grouping×MatMultTuple×Weight [g,m,w] =
          MatrixMultOnGrouping(g1, g2);
 3      ValueTuple v_Tuple = [];
 4      for i ← 1 to |m| do
 5          Value v = ⟨vt1, vt2⟩(m(i));
 6          v_Tuple = v_Tuple || v;
 7      end
 8      Tuple×Tuple [inducedValueTuple, inducedReductionTuple] =
          CollapseClassesLeftmost(v_Tuple);
 9      Grouping×Weight [g, fw] = Reduce(g, inducedReductionTuple,
          v_Tuple);
10      ValueTuple valueTuple = one of [0̄, 1̄], [1̄, 0̄], [0̄], [1̄] based on
          inducedValueTuple ;
11      WCFLOBDD n = RepresentativeCFLOBDD(w · fw · fw1 · fw2, g,
          valueTuple);
12      return n;
13  end
```

where, W is

$$W = \begin{array}{c} weightOfExit^{g.AConnection}_{1\times p} \times \\ \begin{bmatrix} \vdots \\ weightOfExit^{g.BConnections[j]}_{1\times k_j} \\ \times PM^{g.BReturnTuples[j]}_{k_j\times e} \\ \vdots \end{bmatrix}_{\substack{p\times e \\ j\in\{1..p\}}} \end{array}$$

Pseudo-code for the algorithm is given as Alg. 75.

M.5.2 Sampling

To sample an assignment from the probability distribution efficiently, we need to perform the operation directly on the WCFLOBDD that represents the probability

Algorithm 72: MatrixMultOnGrouping

```
    Input: Groupings g1, g2
    Output: Grouping×MatMultTuple×Weight [g,m,w] such that g = g1 ×
            g2
 1  begin
 2  |  if g1.level == 1 then    // Base Case:  matrices of size 2 × 2
    |  |  // Construct a level 1 Grouping that reflects which
    |  |     cells of the product hold equal entries in the
    |  |     output MatMultTuple
 3  |  end
 4  |  if g1 or g2 is ConstantZeroProtoCFLOBDD then
 5  |  |  return [ ConstantZeroProtoCFLOBDD(g1.level), [{[(0,0), 0̄]}], 0̄ ];
 6  |  end
 7  |  if g1 is Identity-Proto-CFLOBDD then
 8  |  |  m = [ {[(0,k), 1̄]} : k ∈ [1..g2.numberOfExits] ];
 9  |  |  return [g2, m, 1̄];
10  |  end
11  |  if g2 is Identity-Proto-CFLOBDD then
12  |  |  m = [ {[(k,0), 1̄]} : k ∈ [1..g1.numberOfExits] ];
13  |  |  return [g1, m, 1̄];
14  |  end
15  |  InternalGrouping g = new InternalGrouping(g1.level);
16  |  Grouping×MatMultTuple×Weight [aa,ma,wa] =
    |   MatixMultOnGrouping(g1.AConnection, g2.AConnection);
17  |  g.AConnection = aa; g.AReturnTuple = [1..|ma|];
18  |  g.numberOfBConnections = |ma|;
    |  // Continued in Alg. 73
```

distribution. Suppose that the WCFLOBDD has l levels. If the distribution were given as a vector of weights, $W = [w_1, \cdots, w_{2^{2^l}}]$, then the probability of selecting the pth matched path would be

$$Prob(p) = \frac{w_p}{\Sigma_{i=1}^{2^{2^l}} w_i}$$

Because we do not have this information directly, instead of sampling one matched path, we will sample a set of matched paths that lead to an exit vertex. At top level, we will consider only those paths that lead to the terminal value $\bar{1}$. At every

Algorithm 73: MatrixMultOnGrouping (cont.)

```
19
     // Interpret ma to (symbolically) multiply and add
        BConnections
20   MatMultTuple m = [];
21   Tuple valueTuple = [];
22   for i ← 1 to |ma| do        // Interpret i^th BP in ma to create
      g.BConnections[i]
        // Set g.BConnections[i] to the (symbolic) weighted dot
           product
           Σ_{((k1,k2),v)∈ma(i)} v * g1.BConnections[k1] * g2.BConnections[k2]
23      CFLOBDD curr_cflobdd = ConstantCFLOBDD(g1.level, [0_BP]);
24      for ((k1, k2), v) ∈ ma(i) do
25          Grouping×MatMultTuple×Weight [bb,mb,wb] =
             MatrixMultOnGrouping(g1.BConnections[k1],
             g2.BConnections[k2]);
26          MatMultTuple mc = [];
27          Tuple inducedValueTuple = [];
28          for j ← 1 to |mb| do
29              BP bp = ⟨g1.BReturnTuples[k1],
                 g2.BReturnTuples[k2]⟩(mb(j));
30              mc = mc || bp; valueTuple_B = valueTuple_B || wb;
31          end
32          Tuple×Tuple [inducedMatMultTuple, inducedReductionTuple]
             = CollapseClassesLeftmost(mc);
33          [bb, fw] = Reduce(bb, inducedReductionTuple, valueTuple_B);
34          CFLOBDD n = RepresentativeCFLOBDD(fw, bb,
             inducedMatMultTuple);
35          curr_cflobdd = curr_cflobdd + v * n ;          // Accumulate
             symbolic sum
36      end
37      g.BConnection[i] = curr_cflobdd.grouping;
38      g.BReturnTuples[i] = curr_cflobdd.valueTuple;
39      m = m || curr_cflobdd.valueTuple;
40      valueTuple = valueTuple || curr_cflobdd.factor_weight
41   end
```

Algorithm 74: MatrixMultOnGrouping (cont.) from Alg. 73

```
19
20      g.numberOfExits = |m|;
21      Tuple×Tuple [inducedMatMultTuple, inducedReductionTuple] =
          CollapseClassesLeftmost(m);
22      [g, fw] = Reduce(g, inducedReductionTuple, valueTuple);
23      return [RepresentativeGrouping(g), m, fw];
24  end
```

Algorithm 75: (WCFLOBDDs) ComputeWeights

```
    Input: Grouping g
1   begin
2       if g.level == 0 then
3           if g == DontCareGrouping then
4               g.weightsOfExits = [g.lw + g.rw];
5           else // g == ForkGrouping
6               g.weightsOfExits = [g.lw, g.rw];
7           end
8       else
9           ComputeWeights(g.AConnection);
10          for i ← 1 to g.numberOfBConnections do
11              ComputeWeights(g.BConnection[i]);
12          end
13          g.weightsOfExits = a 0̄-initialized array of length
              |g.numberOfExits|;
14          for i ← 1 to g.numberOfBConnections do
15              for j ← 1 to g.BConnection[i].numberOfExits do
16                  k = BReturnTuples[i](j);
17                  g.weightsOfExits[k] +=
18                      g.AConnection.weightsOfExits[i] *
                      g.BConnection[i].weightsOfExits[j];
19              end
20          end
21      end
22  end
```

other grouping g, given an exit-vertex e, we will sample a path from all the matched paths that lead to e.

We take advantage of the structure of matched paths to break the assignment/path-sampling problem down into a sequence of smaller assignment/path-sampling problems that can be performed recursively. At each grouping g visited by the algorithm, the goal is to sample a matched path based on the weight of the matched path from the set of matched paths $P_{g,i}$ (in the proto-WCFLOBDD headed by g) that lead from g's entry vertex to a specific exit vertex i of g.

Consider a grouping g and a given exit vertex i. For each middle vertex m of g, the sum of weights of the matched paths from the entry vertex of g to i that passes through m forms a distribution D on g's middle vertices. To sample a matched path from $P_{g,i}$, we (i) first sample the index (m_{index}) of a middle vertex of g according to D, (ii) recursively sample on g.AConnection with respect to the exit vertex that leads to m_{index}, (iii) recursively sample on g.BConnection[m_{index}] with respect to the exit vertex that leads to i, (iv) concatenate the sampled paths to obtain the sampled path of g.

Only those B-connections of g whose exit vertices are connected to i contribute to the paths leading to i. Therefore, to sample a middle vertex, we need to consider only those B-connection groupings that lead to i. For such an i-connected B-connection grouping k, let $(g.\mathit{BReturnTuples}[k])^{-1}[i]$ denote the exit vertex of $g.\mathit{BConnections}[k]$ that leads to i; i.e., $\langle j, i\rangle \in g.\mathit{BReturnTuples}[k] \Leftrightarrow (g.\mathit{BReturnTuples}[k])^{-1}[i] = j$.

Given the sum of weights of all matched paths leading to exit vertex i as *weightsOfExits*[i], we can sample m_{index} based on the following probability espression (where $g.A$ denotes $g.\mathit{AConnection}$, $g.B[k]$ denotes $g.\mathit{BConnections}[k]$, and $g.\mathit{BRT}$ denotes $g.\mathit{BReturnTuples}$):

$$\mathit{Prob}(m_{index}) = \frac{\begin{array}{c}\mathit{weightsOfExit}^{\,g.A}[m_{index}] \\ \times\, \mathit{weightsOfExit}^{\,g.B[m_{index}]}[(g.\mathit{BRT}[m_{index}])^{-1}[i]]\end{array}}{g.\mathit{weightsOfExit}[i]} \tag{M.1}$$

Using this process, m_{index} is selected. The sampling procedure is called recursively on g's A-connection, which returns an assignment a_A, and then on B-connection[m_{index}]

Algorithm 76: Sample an Assignment from a WCFLOBDD

```
1 Algorithm SampleAssignment(n)
    Input: WCFLOBDD n = ⟨fw, g, vt⟩
    Output: Assignment sampled from n according to vt.
2   begin
3       i = the index of 1̄ in vt;
4       Assignment a = SampleOnGroupings(g, i);
5       return a;
6   end
7 end
```

of g, which returns an assignment a_B. The sampled path/assignment of grouping g is $a = a_A || a_B$.

At level-0 (the base case), if g is a `DontCareGrouping`, the assignment is sampled from "0" and "1" in proportion to the edge weights (lw, rw). If g is a `ForkGrouping`, the assignment "0" or "1" is chosen according to the specified index i. Pseudo-code for the sampling algorithm is given as Algs. 76 and 77.

Algorithm 77: Sample an Assignment from a WCFLOBDD

```
1  SubRoutine SampleOnGroupings(g, i)
     Input: Grouping g, Exit index i
     Output: Assignment for a path leading to exit i of g, sampled in
             proportion to path weights
2    begin
3      if g.level == 0 then
4        if g==DontCareGrouping then
           // random(w1,w2) returns 1 if w1 is chosen, else
              w2
5          return random(g.lw, g.rw) ? "0" : "1";
6        else // g==ForkGrouping, so i ∈ [1, 2]
7          return (i == 1) ? "0" : "1"
8        end
9      end
10     Tuple WeightsOfPathsLeadingToI = [];
11     for j ← 1 to g.numberOfBConnections do      // Build weight
        info tuple from which to sample
12       if i ∈ g.BReturnTuples[j] then       // if j^th B-connection
          leads to i
13         WeightsOfPathsLeadingToI = WeightsOfPathsLeadingToI
            || (g.AConnection.WeightsOfExits[j] *
            g.BConnections[j].weightsOfExits[k]), where i =
            BReturnTuples[j](k);
14       end
15     end
16     m_index ← Sample(WeightsOfPathsLeadingToI);           // Sample
        middle-vertex index m_index
17     Assignment a = SampleOnGroupings(g.AConnection, m_index) ||
        SampleOnGroupings(g.BConnection[m_index], k), where i =
        BReturnTuples[m_index](k);
18     return a;
19   end
20 end
```

Bibliography

Parosh Aziz Abdulla, Yo-Ga Chen, Yu-Fang Chen, Lukáš Holík, Ondřej Lengál, Jyun-Ao Lin, Fang-Yi Lo, and Wei-Lun Tsai. Verifying quantum circuits with level-synchronized tree automata. *Proceedings of the ACM on Programming Languages*, 9(POPL):923–953, 2025.

Gadi Aleksandrowicz et al. Qiskit: An open-source framework for quantum computing, 2021.

Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, 2004.

Rajeev Alur and Parthasarathy Madhusudan. Adding nesting structure to words. *Journal of the ACM (JACM)*, 56(3):1–43, 2009.

Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005a. doi: 10.1145/1075382.1075387. URL `https://doi.org/10.1145/1075382.1075387`.

Rajeev Alur, Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *International Colloquium on Automata, Languages, and Programming*, pages 1102–1114. Springer, 2005b.

Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 218–233. Springer-Verlag, 1995.

R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods Syst. Des.*, 10(2/3):171–206, 1997. doi: 10.1023/A:1008699807402. URL https://doi.org/10.1023/A:1008699807402.

Thomas Ball and James R. Larus. Efficient path profiling. In *Proc. of MICRO-29*, December 1996.

Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 97–103. ACM, 2001a. doi: 10.1145/379605.379690. URL https://doi.org/10.1145/379605.379690.

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001b. doi: 10.1007/3-540-45139-0_7. URL https://doi.org/10.1007/3-540-45139-0_7.

Mari-Carmen Banuls, Matthew B Hastings, Frank Verstraete, and J. Ignacio Cirac. Matrix product states for dynamical simulation of infinite chains. *Physical review letters*, 102(24):240603, 2009.

Stephane Beauregard. Circuit for Shor's algorithm using $2n+3$ qubits. *arXiv preprint quant-ph/0205095*, 2002.

Michael Benedikt, Patrice Godefroid, and Thomas W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12,*

2001, Proceedings, pages 652–666, 2001. doi: 10.1007/3-540-48224-5_54. URL https://doi.org/10.1007/3-540-48224-5_54.

T. Bhuvaneswari, V.C. Prasad, Ajay Kumar Singh, and P.W.C. Prasad. Weights binary decision diagram (WBDD) and its application to matrix multiplication. In *2009 Innovative Technologies in Intelligent Systems and Industrial Applications*, pages 470–475. IEEE, 2009.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, 1997.

Robert S Boyer and Warren A Hunt Jr. Function memoization and unique object representation for acl2 functions. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 81–89, 2006.

Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45, 1991.

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(6):677–691, August 1986.

Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. of the 30th ACM/IEEE Design Automation Conf.*, pages 535–541, 1995.

Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):1218–1243, 2023.

Cirq Developers. Cirq, December 2022. URL https://doi.org/10.5281/zenodo.7465577. See the full list of authors at github.com/quantumlib/Cirq/graphs/contributors.

Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Chih-Yuan Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of the 30th ACM/IEEE Design Automation Conf.*, pages 54–60, 1993.

Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. Applications of multi-terminal binary decision diagrams. Technical Report CS-95-160, Carnegie Mellon Univ., School of Comp. Sci., April 1995.

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2008.

Robert L. Constable and David Gries. On classes of program schemata. *SIAM J. Comput.*, 1(1):66–118, 1972. doi: 10.1137/0201006. URL `https://doi.org/10.1137/0201006`.

Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. Abstract domains of affine relations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–73, 2014.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19. ACM, 2006. doi: 10.1145/1159876.1159880. URL `https://doi.org/10.1145/1159876.1159880`.

Alain Finkel, Bernard Willems, and Pierre Wolperr. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33 (2):13–es, 2007.

Austin G. Fowler, Simon J. Devitt, and Lloyd C.L. Hollenberg. Implementation of Shor's algorithm on a linear nearest neighbour qubit array. *arXiv preprint quant-ph/0402196*, 2004.

Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods Syst. Des.*, 10(2/3):149–169, 1997. doi: 10.1023/A:1008647823331. URL `https://doi.org/10.1023/A:1008647823331`.

Stephen J. Garland and David C. Luckham. Program schemes, recursion schemes, and formal languages. *J. Comput. Syst. Sci.*, 7(2):119–160, 1973. doi: 10.1016/S0022-0000(73)80040-6. URL `https://doi.org/10.1016/S0022-0000(73)80040-6`.

Eiichi Goto. Monocopy and associative algorithms in an extended LISP. Technical report, Technical Report TR 74-03, University of Tokyo, 1974.

Johnnie Gray. quimb: A python library for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, 2018. doi: 10.21105/joss.00819.

Younes Guellouma and Hadda Cherroun. Efficient implementation for deterministic finite tree automata minimization. *Journal of computing and information technology*, 24(4):311–322, 2016.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL `https://doi.org/10.1561/2500000010`.

Aarti Gupta. *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction.* PhD thesis, Carnegie Mellon Univ., 1994. Tech. Rep. CMU-CS-94-208.

Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive Boolean functions. In *Proc. of the Int. Conf. on Computer Aided Design*, pages 192–199, November 1993.

Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. A tensor network based decision diagram for representation of quantum circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2020.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.

Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. Logical abstractions for noisy variational quantum algorithm simulation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 456–472, 2021.

Jawahar Jain, James R. Bitner, Magdy S. Abadir, Jacob A. Abraham, and Donald S. Fussell. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Trans. on Comp.*, C-46(11):1230–1245, November 1997.

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.

James Koppel. Version space algebras are acyclic tree automata. *CoRR*, abs/2107.12568, 2021. URL `https://arxiv.org/abs/2107.12568`.

Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of Boolean programs. In *International Conference on Concurrency Theory*, pages 203–217. Springer, 2006.

James R Larus. Whole program paths. *ACM SIGPLAN Notices*, 34(5):259–269, 1999.

Ondrej Lhoták. *Program Analysis Using Binary Decision Diagrams.* PhD thesis, McGill University, 2006.

Zechun Li, Peng Zhang, Yichi Zhang, and Hongkun Yang. NDD: A decision diagram for network verification. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 237–258, 2025.

Richard J. Lipton and Kenneth W. Regan. *Quantum Algorithms via Linear Algebra: A Primer.* MIT Press, 2014.

Nancy A. Lynch and Edward K. Blum. A difference in expressive power between flowcharts and recursion schemes. *Math. Syst. Theory*, 12:205–211, 1979. doi: 10.1007/BF01776573. URL `https://doi.org/10.1007/BF01776573`.

Harry G. Mairson. A simple proof of a theorem of statman. *Theor. Comput. Sci.*, 103(2):387–394, 1992. doi: 10.1016/0304-3975(92)90020-G. URL `https://doi.org/10.1016/0304-3975(92)90020-G`.

K McMillan. Hierarchical representations of discrete functions, with application to model checking, computer aided verification, lncs, 1994.

Wannes Meert and Arthur Choi. PySDD, v0.1. Zenodo, 10.5281/zenodo.1202374, March 2018. URL `https://doi.org/10.5281/zenodo.1202374`.

Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications.* Springer Science & Business Media, 1998.

David Melski. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation.* PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, February 2002. Tech. Rep. 1435.

David Melski and Thomas Reps. Interprocedural path profiling. In *Comp. Construct.*, pages 47–62, 1999.

Donald Michie. *Memo functions: a language feature with" rote-learning" properties.* Edinburgh University, Department of Machine Intelligence and Perception, 1967.

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.

Kengo Nakamura, Shuhei Denzumi, and Masaaki Nishino. Variable shift sdd: a more succinct sentential decision diagram. *arXiv preprint arXiv:2004.02502*, 2020.

C. Nandi, M. Willsey, A. Zhu, Y.R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock. Rewrite rule inference using equality saturation. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2021.

Michael A Nielsen and Isaac L Chuang. Quantum information and quantum computation. *Cambridge: Cambridge University Press*, 2(8):23, 2000.

Philipp Niemann, Robert Wille, D. Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. Qmdds: Efficient quantum function representation and manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(1):86–99, 2016. doi: 10.1109/TCAD.2015.2459034. URL `https://doi.org/10.1109/TCAD.2015.2459034`.

Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In Jack B. Dennis, editor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Woods Hole, Massachusetts, USA, June 2-5, 1970*, pages 119–127. ACM, 1970. doi: 10.1145/1344551.1344563. URL `https://doi.org/10.1145/1344551.1344563`.

O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

Frank Reffel. BDD-nodes can be more expressive. In *Proc. of the Asian Computing Science Conference*, December 1999.

T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97: Int. Logic Programming Symposium*, pages 5–19, Cambridge, MA, 1997. M.I.T.

T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, pages 49–61, 1995.

Tsutomu Sasao and Masahira Fujita, editors. *Representations of Discrete Functions*. Kluwer Acad., 1996.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

M. Sistla, S. Chaudhuri, and T. Reps. Symbolic quantum simulation with Quasimodo. In *Computer Aided Verification (CAV)*, 2023.

Meghana Sistla and Thomas Reps. CFLOBDDs: Context-free-language ordered binary decision diagrams. https://github.com/trishullab/cflobdd, 2022.

Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. CFLOBDDs: Context-free-language ordered binary decision diagrams. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–73, 2024. DOI: 10.1145/3651157.

Fabio Somenzi. CUDD: CU decision diagram package–release 2.4.0. *University of Colorado at Boulder*, 2012.

Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2):243–270, 1997.

Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. Bit-slicing the Hilbert space: Scaling up accurate quantum circuit simulation. In *Design Automation Conference (DAC)*, pages 439–444, 2021. doi: 10.1109/DAC18074.2021.9586191.

Frank Verstraete, Juan J Garcia-Ripoll, and Juan Ignacio Cirac. Matrix product density operators: Simulation of finite-temperature and dissipative systems. *Physical review letters*, 93(20):207204, 2004.

George F. Viamontes, Igor L. Markov, and John P. Hayes. Improving gate-level simulation of quantum circuits. *Quantum Inf. Process.*, 2(5):347–380, 2003. doi: 10.1023/B\%3AQINP.0000022725.70000.4A. URL `https://doi.org/10.1023/B%3AQINP.0000022725.70000.4a`.

George F. Viamontes, Igor L. Markov, and John P. Hayes. High-performance QuIDD-based simulation of quantum circuits. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 1354–1355. IEEE Computer Society, 2004. doi: 10.1109/DATE.2004.1269084. URL `https://doi.org/10.1109/DATE.2004.1269084`.

Guifré Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical review letters*, 91(14):147902, 2003.

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, 2023a. doi: 10.22331/Q-2023-09-11-1108. URL https://doi.org/10.22331/q-2023-09-11-1108.

Lieuwe Vinkhuijzen, Thomas Grurl, Stefan Hillmich, Sebastiaan Brand, Robert Wille, and Alfons Laarman. Efficient implementation of LIMDDs for quantum circuit simulation. In Georgiana Caltais and Christian Schilling, editors, *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*, volume 13872 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2023b. doi: 10.1007/978-3-031-32157-3_1. URL https://doi.org/10.1007/978-3-031-32157-3_1.

Sarma B.K. Vrudhula, Massoud Pedram, and Yung-Te Lai. Edge valued binary decision diagrams. *Representations of Discrete Functions*, pages 109–132, 1996.

Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000. ISBN 0-89871-458-3. URL http://ls2-www.cs.uni-dortmund.de/monographs/bdd/.

J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Prog. Lang. Design and Impl.*, 2004.

J. Whaley, D. Avots, M. Carbin, and M.S. Lam. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems*, 2005.

Robert Wille, Lukas Burgholzer, and Michael Artner. Visualizing decision diagrams for quantum computing. In *Design, Automation and Test in Europe*, 2021.

M. Willsey. Fast and extensible equality saturation with egg, 2021. URL blog.sigplan.org/2021/04/06/equality-saturation-with-egg/.

Kieran Woolfe. *Matrix product operator simulations of quantum algorithms*. PhD thesis, University of Melbourne, School of Physics, 2015.

Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

M. Yannakakis. Graph-theoretic methods in database theory. In *Symp. on Princ. of Database Syst.*, pages 230–242, 1990.

Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 542–558, 2021.

Xusheng Zhi and Thomas Reps. Polynomial bounds of CFLOBDDs against BDDs. *ACM Trans. Program. Lang. Syst.*, 47(2):4:1–4:36, 2025. doi: 10.1145/3716313. URL https://doi.org/10.1145/3716313.

Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 38(5):848–859, 2019. doi: 10.1109/TCAD.2018.2834427. URL https://doi.org/10.1109/TCAD.2018.2834427.

Alwin Zulehner and Robert Wille. *Introducing Design Automation for Quantum Computing*. Springer, 2020.

Alwin Zulehner, Stefan Hillmich, and Robert Wille. How to efficiently handle complex values? Implementing decision diagrams for quantum computing. *International Conference on Computer Aided Design (ICCAD)*, 2019. doi: 10.1109/ICCAD45719.2019.8942057. URL https://doi.org/10.1109/ICCAD45719.2019.8942057.