RESEARCH-ARTICLE

CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams

MEGHANA APARNA SISTLA, The University of Texas at Austin, Austin, TX, United States

SWARAT CHAUDHURI, The University of Texas at Austin, Austin, TX, United States

THOMAS REPS, University of Wisconsin-Madison, Madison, WI, United States

Published: 02 May 2024
Online AM: 04 March 2024
Accepted: 23 February 2024
Revised: 07 December 2023
Received: 12 May 2023

CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams

MEGHANA APARNA SISTLA, The University of Texas at Austin, Austin, USA
SWARAT CHAUDHURI, The University of Texas at Austin, Austin, USA
THOMAS REPS, University of Wisconsin–Madison, Madison, USA

This article presents a new compressed representation of Boolean functions, called *CFLOBDDs* (for Context-Free-Language Ordered Binary Decision Diagrams). They are essentially a plug-compatible alternative to BDDs (Binary Decision Diagrams), and hence are useful for representing certain classes of functions, matrices, graphs, relations, and so forth in a highly compressed fashion. 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*. 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 Greenberger-Horne-Zellinger, 524,288 for Bernstein-Vazirani, 4,194,304 for Deutsch-Jozsa, and 4,096 for Grover's algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

CCS Concepts: • **Computing methodologies** → **Representation of Boolean functions**; • **Software and its engineering** → *Formal software verification*; • **Hardware** → *Quantum computation*; **Simulation and emulation**; *Model checking;*

Additional Key Words and Phrases: Decision diagram, matched paths, best-case double-exponential compression, quantum simulation

ACM Reference Format:
Meghana Aparna Sistla, Swarat Chaudhuri, and Thomas Reps. 2024. CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams. *ACM Trans. Program. Lang. Syst.* 46, 2, Article 7 (May 2024), 82 pages. https://doi.org/10.1145/3651157

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

The work was supported, in part, by a gift from Rajiv and Ritu Batra; by the John Simon Guggenheim Memorial Foundation; by Facebook under a Probability and Programming Research Award; by NSF under grants CCR-9986308 and CCF-2212559; by ONR under contracts N00014-00-1-0607 and N00014-19-1-2318; by the MDA under SBIR contract DASG60-01-P-0048 to GrammaTech Inc., and by an S.N. Bose Scholarship to M.A. Sistla. T. Reps has an ownership interest in GrammaTech Inc., which has licensed elements of the technology reported in this publication.
Authors' addresses: M. A. Sistla and S. Chaudhuri, Computer Science Department, The University of Texas at Austin, 2317 Speedway, Stop D9500, Austin, TX 78712, USA; e-mails: mesistla@utexas.edu, swarat@cs.utexas.edu; T. Reps, Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI, 53706, USA; e-mail: reps@cs.wisc.edu.

ACM 0164-0925/2024/05-ART7
https://doi.org/10.1145/3651157

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, and so on, which are encoded as functions of type (i) or (ii). **Binary Decision Diagrams (BDDs)** [11] 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 **Reduced Ordered BDD (ROBDD)** 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)** [13, 15] or **Algebraic Decision Diagrams (ADDs)** [3]. We will refer to ROBDDs/MTBDDs/ADDs generically as BDDs from here on.

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) [5]. 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 [54] to handle relations over valuations over a Boolean program's Boolean variables [6].
— The Datalog solver `bddbddb`, which uses BDDs as the backing representation of relations, was developed by Whaley and Lam [65, 66] to support a variety of program analyses.
— Lhoták [37] 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.

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 article, 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 ROBDD 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.

CFLOBDDs can represent functions, matrices, graphs, relations, and so forth (using binary- or multi-valued terminals, as appropriate). Even for objects for which double-exponential compression is not achieved, CFLOBDDs may provide better compression than BDDs. Like BDDs, CFLOBDDs are *canonical* (Section 4), and operations are performed on them directly (Section 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 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 twofold. 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 (Section 3.5.1). 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 Figures 3 and 6, presented later.)

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 (Section 10.2.1) and for quantum simulation (Section 10.2.2).

Our work makes the following contributions:

- We introduce a new data structure, called *CFLOBDDs*, for representing functions, matrices, graphs, relations, and other discrete structures in a highly compressed fashion (Section 3). In the best case, a CFLOBDD obtains *double-exponential compression in space*—that is, the CFLOBDD for a Boolean function f is double-exponentially smaller than the decision tree for f.
- We present *algorithms* for creating CFLOBDDs and performing operations on them (Section 6). Most operations have low cost: for many of the functions of 2^k variables for which the CFLOBDD representation is double-exponentially smaller than a decision tree of size 2^{2^k}, the CFLOBDD can be constructed in time $O(k)$ and space $O(k)$. Most unary operations on CFLOBDDs are either constant-time or linear in the size of the argument CFLOBDD. The cost of most binary operations is bounded by the product of (i) the sizes of the two argument CFLOBDDs and (ii) the size of the answer CFLOBDD.
- We show an *exponential gap* between CFLOBDDs and BDDs (Section 8).
- We describe how CFLOBDDs can be used to simulate quantum circuits (Section 9.4).
- We measured the performance of CFLOBDDs and BDDs on synthetic and quantum-simulation benchmarks (Section 10). 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 **Greenberger-Horne-Zeilinger (GHZ)**, 1,024× for **Bernstein-Vazirani (BV)**, 8,192× for **Deutsch-Jozsa (DJ)**, and 128× for Grover's algorithm.

Organization. Section 2 reviews decision trees and BDDs. Section 3 introduces the basic principles underlying CFLOBDDs. Section 4 introduces some additional structural invariants that allow us to establish that each Boolean function has a unique, canonical representation as a CFLOBDD. Section 5 discusses how some standard techniques—hash-consing [23] and function-caching (or *memo functions* [45])—apply to CFLOBDDs. Section 6 presents algorithms for a variety of CFLOBDD operations. Section 7 discusses how to represent matrices and vectors using CFLOBDDs, and how to perform some important operations on them. Section 8 demonstrates an exponential gap between CFLOBDDs and BDDs: the CFLOBDD for a function f can

$$H_2 = \begin{array}{c} \\ x_0 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{array}\!\!\begin{array}{l} y_0 \\ \begin{array}{l} 0 \\ 1 \end{array} \end{array} \qquad H_4 = H_2 \otimes H_2 = \begin{array}{c} \\ x_0 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 0 & 1 \end{array} \\ \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} \end{array}\!\!\begin{array}{l} y_0 \\ \begin{array}{l} 0 \\ 1 \end{array} \end{array} = \begin{array}{c} \\ x_0x_1 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 00 & 01 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \end{array}\!\!\begin{array}{l} y_0y_1 \\ \begin{array}{l} 00 \\ 01 \\ 10 \\ 11 \end{array} \end{array}$$

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

be exponentially smaller than any BDD for f. Section 9 discusses the application of CFLOBDDs to simulating quantum circuits. Section 10 poses two experimental questions and presents the results of experiments on synthetic and quantum-simulation benchmarks. Section 11 discusses related work. Section 12 concludes.

In consultation with the EIC, we have limited the presentation to about 62 pages. Additional material is available in our previous work [59], and citations of the form "[59, Section X.Y]" indicate where omitted details can be found. The three most relevant appendices from that work [59] are included in this article (see Appendices A–C). Appendix D concerns the time complexity of a key subroutine used in several of the CFLOBDD operations.

2 PRELIMINARIES: A FAMILY OF EXAMPLES, BOOLEAN FUNCTIONS, DECISION TREES, AND BDDS

This section presents the set of *Hadamard matrices* $\mathcal{H}$, which recur in Section 3 and Sections 8 through 10. 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 Figure 1 as the base case, where $\otimes$ denotes the Kronecker product.[1] Figure 1 shows H_2 and H_4, the first two matrices in $\mathcal{H}$. The *Kronecker product* of two matrices is defined as

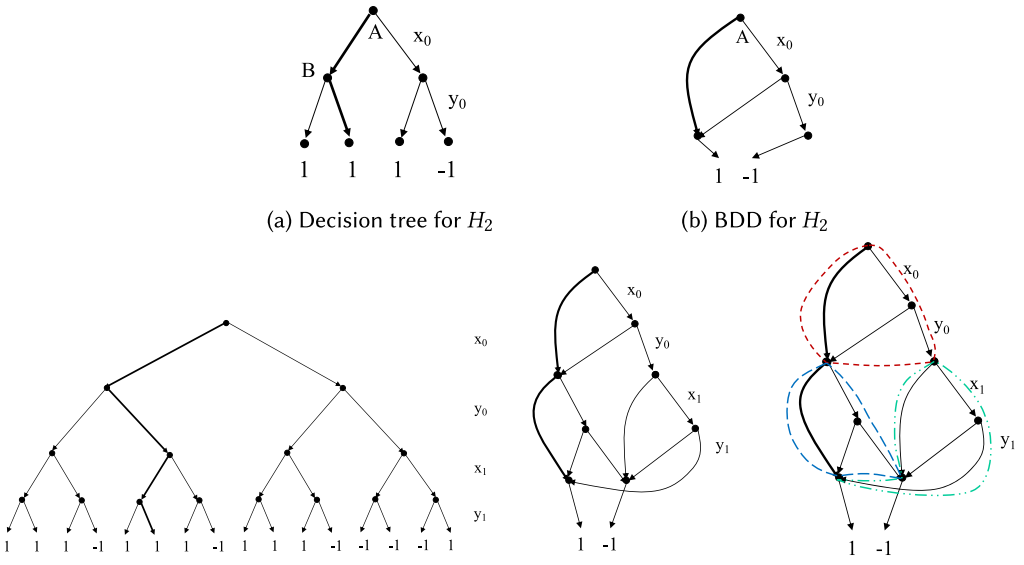
$$A \otimes B = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} \otimes B = \begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{n,1}B & \cdots & a_{n,m}B \end{bmatrix}.$$

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 Figure 1, H_2 requires two variables (x_0 for the row index and y_0 for the column index), whereas H_4 requires four 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 function of type $\{0,1\}^{2^{i-1}} \times \{0,1\}^{2^{i-1}} \to \{-1,1\}$. Our convention is that x_0 and y_0 are the most-significant bits of the row and column indexes, respectively, and x_1 and y_1 are the next-most-significant bits, respectively, and so on.

[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, but we will see in Section 3.4 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

Fig. 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.

Boolean Functions. A *Boolean function* over n variables is a function in $\{F, T\}^n \to \{F, T\}$. This work is also concerned with pseudo-Boolean functions: a *pseudo-Boolean* function over n variables and value domain W is a function in $\{F, T\}^n \to W$. 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.

Hadamard matrix H_{2^i} can be considered to be a (pseudo-)Boolean function in $\{0, 1\}^{2^i} \to \{-1, 1\}$, 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 $\{F, T\}^n \to W$, the decision tree T_B for B is a complete 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 $\langle v_1, \ldots, v_n\rangle$.)

Figure 2(a) and (c) show two decision trees, with the convention that will be used throughout the article 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). Figure 2(a) 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 Figure 2(a), 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. Figure 2(c) 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 Figure 2(c), 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. Figure 2(b) shows the BDD for H_2, using the variable ordering $\langle x_0, y_0 \rangle$. Again, left branches are for F (or 0), and 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 Figure 2(b), 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 $x_0 \mapsto F$.)

Figure 2(d) 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.)

Figure 2(e) 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. Figure 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). 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 earlier, the size of the BDD for H_{2^i} is $O(3^i)$, and hence, compared to decision trees, BDDs achieve *exponential compression* on $\mathcal{H}$.

In contrast, CFLOBDDs employ a different principle than stacking to account for the Kronecker product. Looking ahead, this principle is explained in Section 3.4, and as we will see when we get to Figure 6(b), there is a CFLOBDD of size $O(i)$ that encodes H_{2^i}. Consequently, CFLOBDDs achieve *double-exponential* compression on $\mathcal{H}$. Moreover, in Section 8, we show that this exponential separation is inherent: for every variable ordering, a BDD that represents H_{2^i} requires $\Omega(2^i)$ nodes (Theorem 8.1).

In the remainder of the article, detailed knowledge about BDDs is not essential. The primary purpose of the material that discusses BDDs is to show that CFLOBDDs offer something new, but that material is tangential to being able to understand the CFLOBDD algorithms that we give. The article gives what is essentially a complete account of CFLOBDD operations and invariants, and it is our hope that it could be read by someone who knows little about BDDs. Nevertheless, additional knowledge about BDD internals could help readers appreciate the material in the article. For background about how BDDs are implemented, the reader is referred to the work of Brace et al. [10].

3 CFLOBDDS

CFLOBDDs are a 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* [1]. 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 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, "$\cdot$" denotes logical-and, and "+" denotes logical-or). (See the work of Clark et al. [16, Section 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 $x_{n/2}, \ldots, x_{n-1}$.[2] 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 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.[3] The representation allows sharing across all of 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.

[2]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 Section 4.2 of property (2).)

[3]We are being deliberately vague about how $g_0, g_1, \ldots, g_k$ are combined, because the details are somewhat complicated. See Definitions 3.1 and 4.1 for the precise definition.

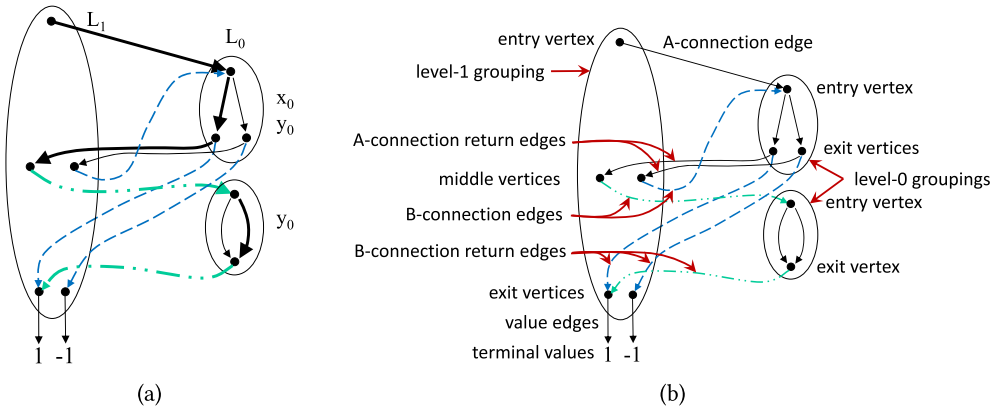


Fig. 3. (a) CFLOBDD for H_2 using the variable ordering $\langle x_0, y_0 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0, 1]$. (b) Guide to the terminology introduced in Definition 3.1.

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 a hierarchical finite-state machine (or, alternatively, the interprocedural control-flow graph for a non-recursive, multi-procedure program).

3.1 Matched Paths

The CFLOBDD representation of H_2 consists of three *groupings*, shown as three ovals in Figure 3(a).[4] 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 Figure 3(a) is called a *fork grouping*.
— A level-0 grouping like the one at the lower right in Figure 3(a) is called a *don't-care grouping*.

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 Figure 3(a) 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.

Figure 3(a) 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

[4]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."

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/outgoing edges of a grouping, such as the pairs of (i) black solid edges and (ii) green dashed-double-dotted edges in the bold path in Figure 3(a), 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 [69]. (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* if and only if the path's word is in the language $L(matched)$ of balanced-parenthesis strings generated by

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

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 figures in the article, 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 reuse 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 they are used to interpret *different variables at different places in a matched path through a CFLOBDD*. For example, the level-0 fork grouping in Figure 3(a) 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 $x_0 \mapsto T$).[5] 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 Figure 3(a), the fork grouping is labeled with x_0 and y_0, and the don't-care grouping with y_0. In general, however, the level-0 groupings interpret *different variables at different places in a matched path*, in accordance with the following principle:

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 n^{th} level-0 grouping visited along a matched path is used to interpret the n^{th} Boolean variable.*

The reader might be worried by the fact that Figure 3(a) contains cycles. In other words, if one ignores the ovals in Figure 3(a), as well as the distinctions among solid, dashed, and

[5] The term *call* is by analogy with how matched paths model the actions of procedure calls in graphs used for interprocedural dataflow analysis [55, 58], interprocedural slicing [30], and model checking hierarchical state machines [9, Section 5].

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 not a matched path and is excluded from consideration.

3.2 CFLOBDD Requirements

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

(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) through (3). To satisfy these more stringent requirements, we define a data structure that is quite different from BDDs (see Sections 3.3 and 4.1). Requirements (1) and (3) are established in Sections 3.3.4 and 3.5.2, respectively. Requirements (2) and (4) are established in Section 4 and Appendix C. Requirement (5) is addressed in Section 5, Section 6, and Appendix D; in particular, Table 1 at the beginning of Section 6 lists the 14 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. Appendix D 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.

3.3 CFLOBDDs Defined, Part I: Basic Structure

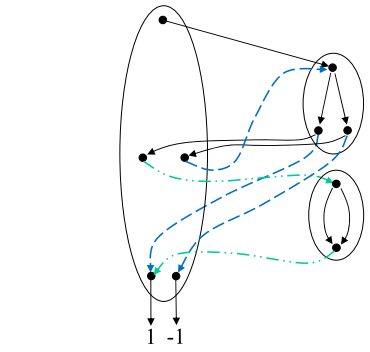
Our formal definition of CFLOBDDs is given in two parts: Definition 3.1 (in the following) and Definition 4.1 (in Section 4.1). Definition 3.1 defines the basic structure of CFLOBDDs, whose various elements are depicted in Figure 3(b). Definition 4.1 imposes some additional structural invariants to ensure that CFLOBDDs provide a canonical representation of Boolean functions. Much about CFLOBDDs can be understood just from Definition 3.1, so we postpone introducing the structural invariants until we address canonicity in Section 4. Where necessary, we distinguish between *mock-CFLOBDDs* (Definition 3.1) and *CFLOBDDs* (Definition 4.1), although we typically drop the qualifier "mock-" when there is little danger of confusion. Figure 3(b) illustrates Definition 3.1 using the CFLOBDD that represents Hadamard matrix H_2.

Definition 3.1 (Mock-CFLOBDD; see Figure 3(b)). 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. A grouping is a collection of vertices and edges (to other groupings), with the structure described in the following.

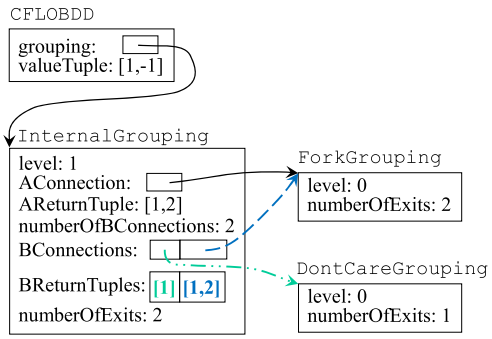
Each grouping g_i at level $0 \leq i \leq k$ has a unique *entry vertex*, which is disjoint from g_i's non-empty set of *exit vertices*.

```
abstract class Grouping {
  level: int
  numberOfExits: int
}
class InternalGrouping extends Grouping {
  AConnection: Grouping
  AReturnTuple: ReturnTuple
  numberOfBConnections: int
  BConnections:
    array[1..numberOfBConnections] of Grouping
  BReturnTuples:
    array[1..numberOfBConnections] of ReturnTuple
}
class DontCareGrouping extends Grouping {
  level = 0
  numberOfExits = 1
}
class ForkGrouping extends Grouping {
  level = 0
  numberOfExits = 2
}
class CFLOBDD {      // Multi-terminal CFLOBDD
  grouping: Grouping
  valueTuple: ValueTuple
}
```

(a)



(b)



(c)

Fig. 4. (a) Datatypes for Grouping, InternalGrouping, DontCareGrouping, ForkGrouping, and CFLOBDD. (b) The CFLOBDD for H_2 (repeated from Figure 3(a)). (c) An instance of class `CFLOBDD` that represents H_2.

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 Figure 3(b), 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*.

3.3.1 An Object-Oriented Pseudo-Code. In later parts of the article, we state algorithms using an object-oriented pseudo-code. In accordance with the terminology introduced previously, the basic classes that are used for representing multi-terminal CFLOBDDs are defined in Figure 4(a):

`Grouping`, `InternalGrouping`, `DontCareGrouping`, `ForkGrouping`, and `CFLOBDD`. More details about the notation used in our pseudo-code can be found in Appendix A.

Figure 4(c) shows how the CFLOBDD from Figure 3(a) 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 Figure 4(c), 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`.

3.3.2 Rationale. The rationale behind the terminology introduced in Definition 3.1, Figure 3(b), and Figure 4(a) 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 (see Figure 4(a)) in the following order, which mimics the form of the grammar for matched paths from Equation (1):

$$\begin{array}{rcl} \textit{matched}(\text{at level } i) & = & \text{AConnection} \quad \textit{matched}(\text{at level } i\text{-}1) \quad \text{AReturnTuple}[\cdot] \\ & & \text{BConnections} \quad \textit{matched}(\text{at level } i\text{-}1) \quad \text{BReturnTuples}[\cdot]. \end{array} \tag{2}$$

3.3.3 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—that is, the entry vertex of a level-0 grouping. In particular, $D(i)$ is described by the following recurrence relation:

$$D(0) = 1 \qquad D(i) = D(i-1) + D(i-1), \tag{3}$$

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

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 in the range [1..g.numberOfExits]
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
```

3.3.4 Soundness and an Operational Semantics. Equation (3) allows us to establish Requirement (1) from Section 3.2. Equation (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 $\{T, F\}^{2^k} \rightarrow V$—that is, 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 Algorithm 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 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 Section 3.5.2.) 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.

3.3.5 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. Figure 3(a) is a multi-terminal CFLOBDD; the level-1 grouping has two exit vertices that are mapped to 1 and −1.

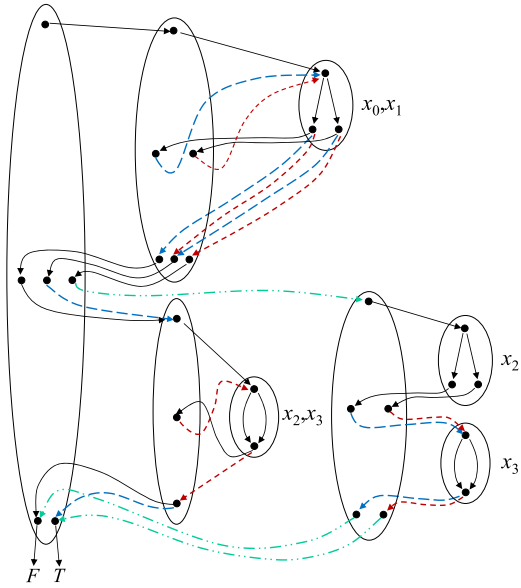


Fig. 5. CFLOBDD for the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. (For clarity, some of the level-0 groupings have been duplicated.)

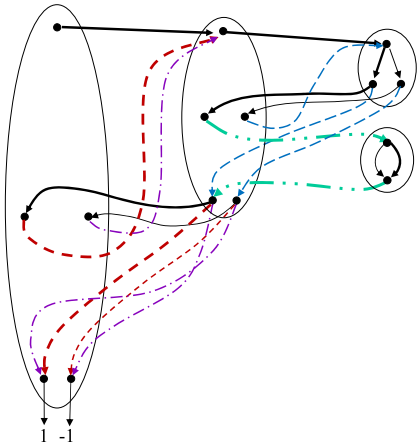
Groupings that have more than two exit vertices naturally arise in the interior groupings of CFLOBDDs—even in Boolean-valued CFLOBDDs. For instance, a level-i−1 grouping used as an A-connection can have more than two exit vertices, in which case the "calling" level-i grouping would have more than two middle vertices. Such multi-terminal groupings can arise in both A-connections and B-connections. Figure 5 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.

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

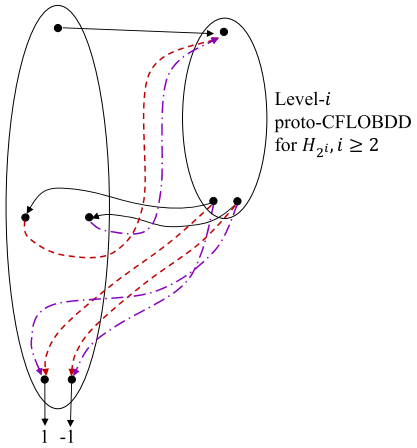
Figure 6(a) shows the CFLOBDD representation of Hadamard matrix H_4 with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. In H_4, the level-1 proto-CFLOBDD is identical to the level-1 proto-CFLOBDD in H_2 (cf. Figures 3(a) and 6(a)). Moreover, in H_4, the A-connection call and both B-connection calls are to the level-1 H_2 lookalike.

Consider how Figure 6(a) 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—that is, 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 Figure 2(e)), the ability of CFLOBDDs to reuse parts of a data structure via a "call" means that there is only a



(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.

Fig. 6. 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 $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$.

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}, $i \geq 2$, $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$ is created by introducing a new outermost grouping at level $i + 1$, again with five vertices and nine edges. (See Figure 6(b).) 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, we have the following principle:

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.*

3.5 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.

3.5.1 Growth of Number of Paths with Level. Let $P(i)$ be the number of matched paths in a proto-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 j^{th} middle vertex of g. Thus, $P(i)$ satisfies the following recurrence equation:

$$P(0) = 2 \qquad P(i) = \sum_j A_j(i-1) \cdot P(i-1). \tag{4}$$

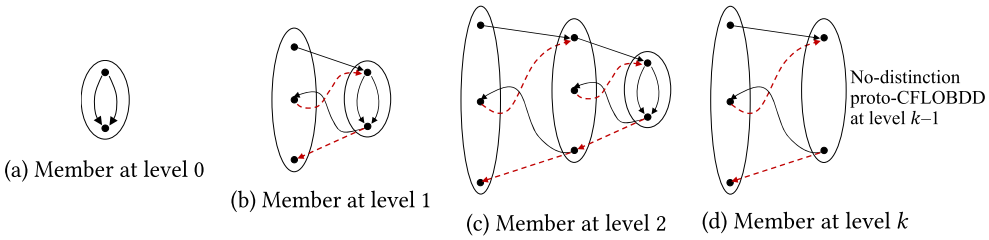


Fig. 7. The family of no-distinction proto-CFLOBDDs.

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 Equation (4) can be rewritten as $P(i) = P(i-1) \cdot P(i-1)$, which has the solution $P(i) = 2^{2^i}$:

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

3.5.2 Best-Case Compression: No-Distinction Proto-CFLOBDDs. Figure 7(a) through (c) 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-CFLOBDD*s. Figure 7(d) 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 vars.	Number of paths	Decision trees			CFLOBDDs (best case)			
		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 Section 3.2 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 ROBDDs [10, 11], 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 Algorithm 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 [64, Theorem 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 Section 8, we show that the CFLOBDD for a function g can be exponentially smaller than *any* ROBDD for g.

3.5.3 Asymptotic Best-Case Compression. Consider a family of functions $F = \{f_j \mid j \geq 0\}$, where the j^{th} 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.[6]
(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 $g_{i\text{-}1}$.[7]

In such a case, the CFLOBDD for each f_i is double-exponentially smaller than the decision tree for f_i—that is, of size $O(i)$ rather than $O(2^{2^i})$. As shown in Figure 6(b), the family of Hadamard matrices $\mathcal{H}$ meets the preceding conditions.

Moreover, in all cases encountered to date, it is possible to give an explicit algorithm for constructing the i^{th} 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

[6]The family of no-distinction proto-CFLOBDDs from Figure 7 is one such family G.
[7]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 $G_1, G_2, \ldots, G_l$ that each grow linearly in the level i.

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.

4 CANONICITY

In this section, we impose some further structural restrictions on proto-CFLOBDDs and CFLOBDDs that go beyond the ideas illustrated earlier (Section 4.1). We then discuss how to establish that CFLOBDDs are a canonical representation of Boolean functions (Section 4.2 and Appendix C).

4.1 CFLOBDDs Defined, Part II: Additional Structural Invariants

As described in Section 3, 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 Figure 4). 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 i^{th} entry of the tuple indicates the element that the i^{th} 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 Figure 5,

- The return tuple associated with the first B-connection of the upper level-1 grouping is $[1, 2]$.
- The return tuple associated with the second 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 Definition 4.1, 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 Section 4.2 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 4.1 (Proto-CFLOBDD and CFLOBDD). A *proto-CFLOBDD* n is a mock-proto-CFLOBDD (Definitions 3.1 and 3.2) in which every grouping/proto-CFLOBDD in n satisfies the *structural*

invariants given in the following. 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 $[1, 2, \ldots, k]$. (In other words, 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. (In other words, 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. (In other words, 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) subsequence 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.[8]

(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.

Figure 8 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.

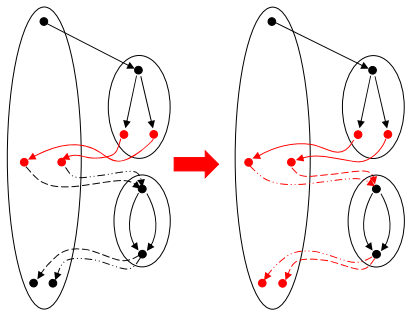
The CFLOBDD from Figure 5 also illustrates the structural invariants:

— 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.

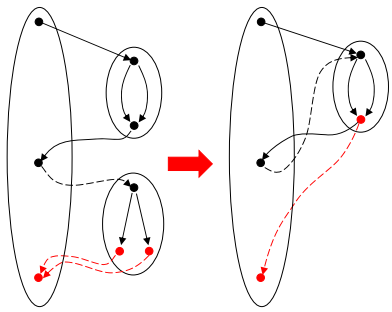
  In the case of rt_1, the set S mentioned in structural invariant 2b is empty; therefore, $n = 0$ and rt_1 is constrained by structural invariant 2b to be $[1, 2]$.

[8]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. Section 5.1 discusses how hash-consing [23] 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 Section 4.2—we say that such structures are "isomorphic" or "equal (up to isomorphism)."

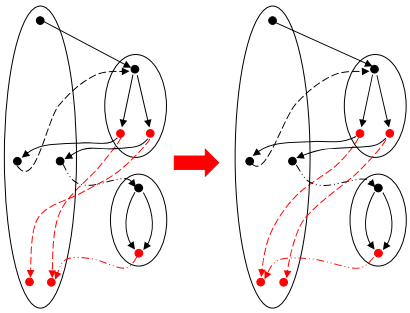
To reduce clutter, our diagrams often show multiple instances of the two kinds of level-0 groupings; in fact, a CFLOBDD can contain at most one copy of each.



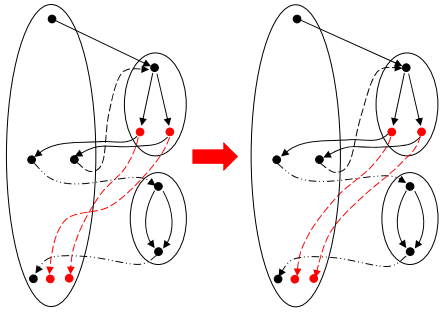
(a) Structural invariant 1



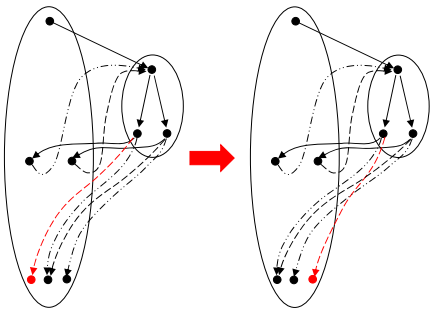
(b) Structural invariant 2a



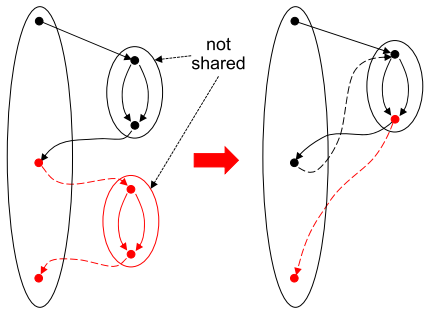
(c) Structural invariant 2b



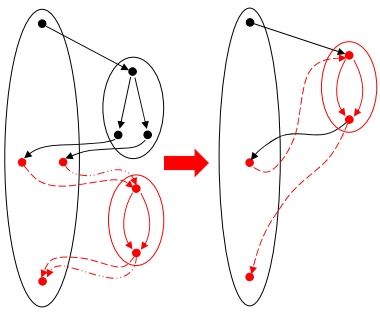
(d) Another case of structural invariant 2b



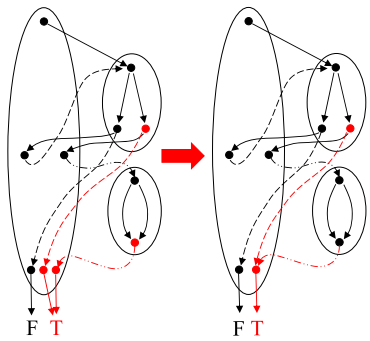
(e) Another case of structural invariant 2b



(f) Structural invariant 3



(g) Structural invariant 4



(h) Structural invariant 6

Fig. 8. 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.

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 Figure 5, 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 invariant 1 to map in order over to the three middle vertices of the level-2 grouping—that is, 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 Section 4.2 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 4.1 (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 Proposition 4.1 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, Proposition 4.1 elucidates a semantic consequence of the structural invariants.[9]

Example 4.2. Proposition 4.1 can be illustrated using Figure 5. If we use numbers to identify exit vertices, ex_C for any grouping g is the sequence $[1..g.\text{numberOfExits}]$. In the upper level-1 grouping in Figure 5, 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 $[1, 2]$.

4.2 Canonicity of CFLOBDDs

CFLOBDDs are a canonical representation of functions over Boolean arguments—that is, 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 8.)

Theorem 4.3 (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 Observation 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

[9]Section 4.2 gives a high-level overview of the proof that CFLOBDDs are a canonical representation of Boolean functions. In the proof of canonicity in Section C, Proposition 4.1 is used in the proof of Proposition C.1, which establishes property (3) from Section 4.2.

— the CFLOBDD for B, again using variable ordering Ord.

By Observation 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 Section 3.3.4.

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 Proposition C.1.)

Note that Obligations 1 and 2 are exactly Requirements (1) and (2) from Section 3.2, respectively. Moreover, Obligations 1 through 3 together show that Requirement (4) from Section 3.2 is met.

5 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.

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

Hash-consing [23] enforces the invariant that only a single representative exists for each value constructed from some datatype. Hash-consing should not be confused with canonicity (Section 4.2 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[10] satisfies an even stronger form of equivalence. In particular, Theorem 4.3 can be restated to read "... then C_1 and C_2 are identical."

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

Because the operations that construct Groupings and CFLOBDDs 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 Groupings and CFLOBDDs, respectively. For instance, a call RepresentativeGrouping(g) checks to see whether a representative for g is already in the table of representative Groupings; if there is such a representative, say h, then g is discarded and h is returned as the result; if there is no such representative, 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 Section 6 that create InternalGroupings, such as PairProduct (Algorithm 9) and Reduce (Algorithm 10), 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 (Algorithm 3), which constructs the members of the family of no-distinction proto-CFLOBDDs depicted in Figure 7, 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 (Algorithm 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—that is, structural invariants (1) through (4) of Definition 4.1 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 Algorithm 10 of Section 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.

5.2 Equality Testing for CFLOBDDs and Proto-CFLOBDDs

As discussed in Section 5.1, 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 Groupings 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 Definition 4.1.

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 Algorithm 1, line 10, we saw how testing whether the argument g is a NoDistinctionProtoCFLOBDD allows further recursive calls to `InterpretGrouping()` to be short-circuited.

5.3 Function Caching

A function cache (or *memo function* [45]) 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 article, such as `PairProduct` (Algorithm 9) and `Reduce` (Algorithm 10). 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 (Section 5.2) also improves the performance of function caching.

6 ALGORITHMS ON CFLOBDDS

In this section and Section 7, we describe operations to construct or combine CFLOBDDs. To aid the reader, Table 1 lists the 14 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 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 (Definition 4.1).

6.1 Primitive CFLOBDD-Creation Operations

6.1.1 Constant Functions. The CFLOBDD-creation operation `ConstantCFLOBDD`, given as lines 1 through 5 of Algorithm 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` (Algorithm 3), which constructs the no-distinction proto-CFLOBDD for a given level k (see also Figure 7). `ConstantCFLOBDD` can be used to construct CFLOBDDs for the constant functions $\lambda x_0, x_1, \ldots, x_{2^k-1}.F$ and $\lambda x_0, x_1, \ldots, x_{2^k-1}.T$ (lines 6–10 and 11–15 of Algorithm 2, respectively). `ConstantCFLOBDD`(k, v) runs in time $O(k)$ and uses at most $O(k)$ space.

Table 1. List of Operations on CFLOBDDs

Operation	Type Signature	Description	Time Complexity	
			CFLOBDD	BDD
Equal (Section 5.2)	CFLOBDD × CFLOBDD → Boolean	Checks if two CFLOBDDs are equal	$O(1)$	$O(1)$
ConstantCFLOBDD (Algorithm 2, Section 6.1.1)	Int(k) × Value(v) → CFLOBDD	Creates a CFLOBDD for a constant function $\lambda x_0 \ldots x_{2^k-1}.v$	$O(k)$	$O(2^k)$
FalseCFLOBDD (Algorithm 2, Section 6.1.1)	Int (k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.F$	$O(k)$	$O(2^k)$
TrueCFLOBDD (Algorithm 2, Section 6.1.1)	Int (k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.T$	$O(k)$	$O(2^k)$
NoDistinctionProtoCFLOBDD (Algorithm 3, Section 6.1.1)	Int (k) → Proto-CFLOBDD	Creates a NoDistinctionProtoCFLOBDD for 2^k variables	$O(k)$	N/A
ProjectionCFLOBDD (Algorithm 4, Section 6.1.2)	Int(k) × Int(i) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \ldots x_{2^k-1}.x_i$	$O(k)$	$O(2^k)$
FlipValueTupleCFLOBDD (Algorithm 5, Section 6.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are flipped	$O(1)$	$O(\|c\|_B)$
ComplementCFLOBDD (Algorithm 5, Section 6.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are complemented	$O(1)$	$O(\|c\|_B)$
ScalarMultiplyCFLOBDD (Algorithm 6, Section 6.2.2)	CFLOBDD (c) × Value (v) → CFLOBDD	Performs $c' = c * v$	$O(\|c\| \times \|c'\|)$	$O(\|c\|_B)$
BinaryApplyAndReduce (Algorithm 8, Section 6.3)	CFLOBDD (c_1) × CFLOBDD (c_2) × Operation op → CFLOBDD	Performs $c' = c_1\ op\ c_2$	$O(\|c_1\| \times \|c_2\| \times \|c'\|)$	$O(\|c_1\|_B \times \|c_2\|_B)$
PathCounting ([59, Algorithm 21], Section 6.4.1)	CFLOBDD(c) → CFLOBDD	Computes the number of paths to every exit vertex of every grouping	$O(\|c\|)$	$O(\|c\|_B)$ (See [59, Section 10.1.2].)
Sampling ([59, Algorithm 22], Section 6.4.2)	CFLOBDD(c) → String	Samples a path from c	$O(\max(vars, \|c\|))$	$O(\max(vars, \|c\|_B))$ (See [59, Section 10.1.2].)
KroneckerProduct ([59, App. G & Algorithm 17], Section 7.2)	CFLOBDD(c_1) × CFLOBDD(c_2) → CFLOBDD	Performs $c' = c_1 \otimes c_2$	$O((\|c_1\| + \|c_2\| + c_1.\#\text{exits} \times c_2.\#\text{exits}) \times \|c'\|)$	$O(\|c_1\|_B)$
MatrixMultiply ([59, Algorithms 19 & 20], Section 7.3)	CFLOBDD(c_1) × CFLOBDD(c_2) → CFLOBDD	Performs $c' = c_1 \times c_2$ for matrices of size $N \times N$	$O(N^3)$, plus the time for a final call to Reduce	$O(N^3)$

Note: *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 $O(2^k)$. 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

```
1  Algorithm ConstantCFLOBDD(k,v)
       Input: int k, Value v
       Output: CFLOBDD representation of the constant function λx_0, x_1, ..., x_{2^k-1}.v
2      begin
3          return RepresentativeCFLOBDD(NoDistinctionProtoCFLOBDD(k), [v]);
4      end
5  end
6  Algorithm FalseCFLOBDD(k)
       Input: int k
       Output: CFLOBDD representation of the constant function λx_0, x_1, ..., x_{2^k-1}.F
7      begin
8          return ConstantCFLOBDD(k, F);
9      end
10 end
11 Algorithm TrueCFLOBDD(k)
       Input: int k
       Output: CFLOBDD representation of the constant function λx_0, x_1, ..., x_{2^k-1}.T
12     begin
13         return ConstantCFLOBDD(k, T);
14     end
15 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
```

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 $2^k - 1$. Figure 9 illustrates the structure of the CFLOBDDs that represent these functions. Algorithm 4 gives pseudo-code for `ProjectionCFLOBDD`(k, i), which constructs the i^{th} such function. `ProjectionCFLOBDD`(k, i) runs in time $O(k)$ and uses at most $O(k)$ space.

6.2 Unary Operations on CFLOBDDs

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

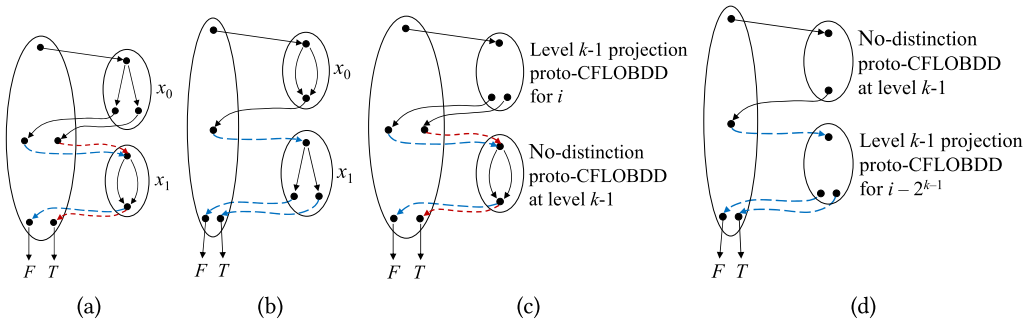


Fig. 9. (a) CFLOBDD for $\lambda x_0 x_1 . x_0$. (b) CFLOBDD for $\lambda x_0 x_1 . x_1$. (c) Schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1} . x_i$, when $0 \leq i < 2^{k-1}$. (d) Schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \ldots, x_{2^k-1} . x_i$, when $2^{k-1} \leq i < 2^k$.

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 CFLOBDD'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 pseudo-code for these functions is given in Algorithm 5. `FlipValueTupleCFLOBDD` and `ComplementCFLOBDD` are constant-time operations.

6.2.2 Scalar Multiplication. Function `ScalarMultiplyCFLOBDD` of Algorithm 6 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 Section 6.3)—with the multiplication operator Times passed as the third argument.

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. In other words, 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 $n_1 \,\mathsf{op}\, n_2$ where, for each assignment a, $(n_1 \,\mathsf{op}\, n_2)(a) = n_1(a) \,\mathsf{op}\, n_2(a)$. Operation op could be $+, -, *, /$, and so forth, or—if the functions are Boolean-valued—$\vee, \wedge, \oplus$, and so forth. As with BDDs, such operations on CFLOBDDs can be implemented via a two-step process:[11]

(1) perform a product construction;
(2) perform a reduction step on the result of step 1.

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 (Section 5.1) and function

[11]The two-step process is conceptual for BDDs: the two steps can be combined in an implementation (e.g., see [19, Section 3.3]). For CFLOBDDs, it does not appear possible to combine the two steps, at least not easily. For more details, see the Remark just after Example 6.1.

ALGORITHM 4: 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
7  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
8      begin
9          if k == 0 then                                          // i must also be 0
10             return RepresentativeForkGrouping;
11         else
12             InternalGrouping g = new InternalGrouping(k);
13             if i < 2**(k-1) then                          // i falls in AConnection range
14                 g.AConnection = ProjectionProtoCFLOBDD(k-1,i);
15                 g.AReturnTuple = [1,2];
16                 g.numBConnections = 2;
17                 g.BConnection[1] = NoDistinctionProtoCFLOBDD(k-1);
18                 g.BReturnTuples[2] = [1];
19                 g.BConnections[2] = g.BConnection[1];
20                 g.BReturnTuples[2] = [2];
21                 g.numberOfExits = 2;
22             else                                          // i falls in BConnection range
23                 g.AConnection = NoDistinctionProtoCFLOBDD(k-1);
24                 g.AReturnTuple = [1];
25                 g.numBConnections = 1;
26                 i = i - 2**(k-1);                  // Remove high-order bit for recursive call
27                 g.BConnections[1] = ProjectionProtoCFLOBDD(k-1,i);
28                 g.BReturnTuples[1] = [1,2];
29                 g.numberOfExits = 2;
30             end
31             return RepresentativeGrouping(g);
32         end
33     end
34 end
```

caching (Section 5.3) 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—that is, $O(|n_1| \times |n_2|)$.[12] Appendix D 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 $n_1 \,\mathsf{op}\, n_2$. Consequently, binary operations

[12]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 5: 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 6: ScalarMultiplyCFLOBDD

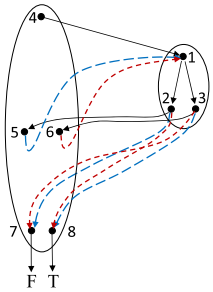
```
   Input: CFLOBDD c, Value v
   Output: CFLOBDD c′ = c ∗ v
1  begin
       // Multiply CFLOBDD c by the CFLOBDD for the constant function λx_0, x_1, ..., x_{2^k−1}.v
2      return BinaryApplyAndReduce(c, ConstantCFLOBDD(c.level, v), (op)Times);          // (See
        Section 6.3)
3  end
```

satisfy Requirement (5)—that is, they run in (expected) time that is polynomial in the sizes of the input and output CFLOBDDs.

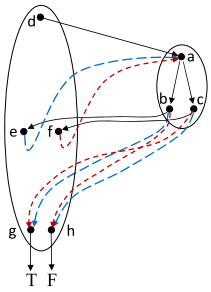
Figure 10 illustrates this method by showing how the CFLOBDD for $\lambda x_0, x_1.T$ is obtained as the result of a Boolean-$\vee$: $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$. Figure 10(c) shows the result of the product construction (`PairProduct`, Algorithm 9). Figures 10(d) through (g) illustrate some of the steps of the reduction algorithm (`Reduce`, Algorithm 10). Figure 10 is discussed in more detail in Example 6.1.

The algorithms involved are given as Algorithms 7 through 11. In Algorithms 8 and 9, we assume that the `CFLOBDD` or `Grouping` arguments are objects whose highest-level groupings are all at the same level:

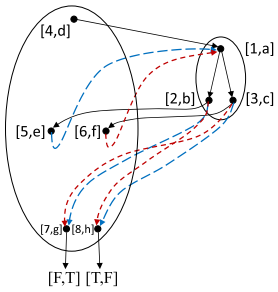
— The operation `BinaryApplyAndReduce`, given as Algorithm 8, starts with a call on `PairProduct` (line 2). `PairProduct`, given as Algorithm 9, performs a recursive traversal of the two `Grouping` arguments, `g1` and `g2`, to create a proto-CFLOBDD that represents a kind of cross product. `PairProduct` returns `g`, 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 Algorithm 9, lines 2–5 and lines 11–29.)



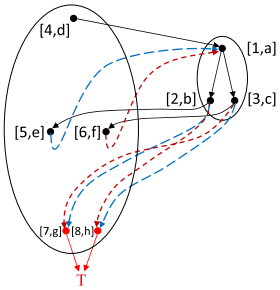
(a) $\lambda x_0, x_1.x_0 \oplus x_1$



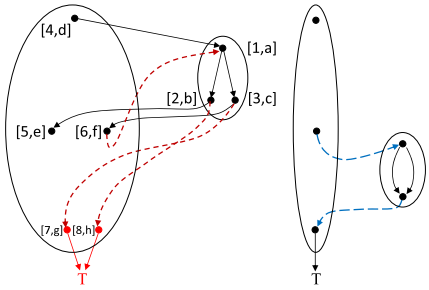
(b) $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$



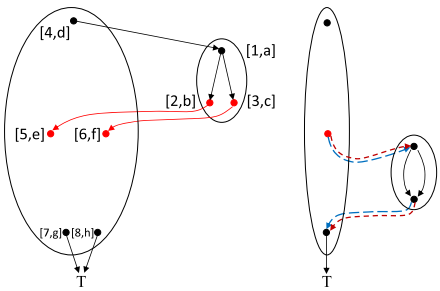
(c) Result of calling `PairProduct` on (a) and (b)



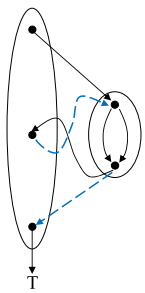
(d) Result of applying $\vee$ to the values in each of the terminal-value pairs $[F, T]$ and $[T, F]$. At this point, it is necessary to perform a reduction that folds together the two exit vertices.



(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 $\lambda x_0, x_1.T$.

Fig. 10. Illustration of how $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$ results in $\lambda x_0, x_1.T$.

ALGORITHM 7: 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 8: 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
```

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 (Section 5.3) is performed for PairProduct. Consequently, for a given invocation of BinaryApplyAndReduce on CFLOBDDs n_1 and n_2, for 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 25). 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

ALGORITHM 9: 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| ;
       // Pair the B-connections, but only for pairs in ptA
       // Descriptor of pairings of exit vertices
11     Tuple ptAns = [];
       // Create a B-connection for each middle vertex
12     for j ← 1 to |ptA| do
13         Grouping×PairTuple [gB,ptB] = PairProduct(g1.BConnections[ptA(j)(1)],
             g2.BConnections[ptA(j)(2)]);
14         g.BConnections[j] = gB ;
           // Now create g.BReturnTuples[j], and augment ptAns as necessary
15         g.BReturnTuples[j] = [] ;
16         for i ← 1 to |ptB| do
17             c1 = g1.BReturnTuples[ptA(j)(1)](ptB(i)(1));          // an exit vertex of g1
18             c2 = g2.BReturnTuples[ptA(j)(2)](ptB(i)(2)) ;         // an exit vertex of g2
19             if [c1,c2] ∈ ptAns then                    // Not a new exit vertex of g
20                 index = the k such that ptAns(k) == [c1,c2] ;
21                 g.BReturnTuples[j] = g.BReturnTuples[j] || index ;
22             else                                       // Identified a new exit vertex of g
23                 g.numberOfExits = g.numberOfExits + 1 ;
24                 g.BReturnTuples[j] = g.BReturnTuples[j] || g.numberOfExits ;
25                 ptAns = ptAns || [c1,c2] ;
26             end
27         end
28     end
29     return [RepresentativeGrouping(g), ptAns];
30 end
```

line 2 of Algorithm 8 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 12).

Lines 2 through 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.

ALGORITHM 10: 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 11: 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 occurence 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
```

— `BinaryApplyAndReduce` then uses `pt`, together with `op` and the value tuples from `CFLOBDDs` `n1` and `n2`, to create the tuple `deducedValueTuple` of leaf values that should be associated with the exit vertices (see Algorithm 8, line 3).

However, `deducedValueTuple` is a *tentative* value tuple for the constructed `CFLOBDD`; because of structural invariant 6 of Definition 4.1, 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` (Algorithm 7):

– 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 of Definition 4.1.

– 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 Algorithm 8, line 4, as well as Algorithm 7.)

— 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` (Algorithm 8, line 5).

Procedure `Reduce`, given as Algorithm 10, recursively traverses `Grouping g`, working in the backward 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 Algorithm 10, 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 (Algorithm 10, 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` (Algorithm 11) in line 15 of `Reduce` enforces structural invariant 4 of Definition 4.1.[13]
- As the B-connections are processed, `Reduce` uses the `position` information returned from `InsertBConnection` to build up the tuple `reductionTupleA` (Algorithm 10, 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 Algorithm 10, lines 18–21.)

Function caching (Section 5.3) is performed for `Reduce`, with respect to both arguments `g` and `reductionTuple`. Appendix D 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 `Groupings`. We do not want 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 invariant 4 of Definition 4.1—and hence why it is permissible to call `RepresentativeGrouping(g)` in line 29 of Algorithm 9. The proof can be found in our prior work [59, Appendix D].

Last, 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 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 6.1. Figure 10 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 $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$. Figure 10(c) 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 (Figure 10(e) and (f)), which would be discovered by `InsertBConnection` (Algorithm 11), 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 $\lambda x_0, x_1.T$ (see Figure 10(g)).

For another example that illustrates `Reduce`, see Example D.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 [19, Section 3.3].

[13] 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.

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` (Algorithm 10) follows the sequence of elements of a matched path backward. `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 `Groupings` that appear in the hash-consing tables are the heads of fully fledged proto-CFLOBDDs, not mock-proto-CFLOBDDs—that is, structural invariants 1 through 4 of Definition 4.1 hold. While such `Groupings` 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. Details about how to perform ternary operations can be found in our previous work [59, 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, of that work [59].

6.4 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 (Section 6.4.1). With those numbers in hand, it is then possible to sample an assignment/path according to the distribution that the CFLOBDD represents (Section 6.4.2).

The same approach can be used for CFLOBDDs whose terminal values are complex numbers, except that the weight of a matched path is the square of the terminal value's absolute value. This approach is used in the application of CFLOBDDs to quantum simulation (Sections 9 and 10.2.2).

6.4.1 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 i^{th} element is the number of matched paths from g's entry vertex to the i^{th} 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 j^{th} row is the vector of path-counts from the j^{th} 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 j^{th} 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.

This definition can be stated equationally, where the expression in large brackets represents M_B.

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

Example 6.2. For the five proto-CFLOBDDs depicted in Figure 11, the vectors of path-counts are computed as follows (read top-to-bottom by level):

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

Pseudo-code for the path-counting algorithm can be found in our previous work [59, Algorithm 21].

6.4.2 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 Section 3.3.4, we assume that an assignment is an array of Booleans, whose entries—starting at index-position 1—are the values of successive variables. Suppose that the CFLOBDD has l levels. If the distribution were 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 p^{th} matched path is given by

$$Prob(p) = \frac{w_p}{\sum_{i=1}^{2^{2^l}} w_i}. \tag{5}$$

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 Section 6.4.1 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 Equation (5) 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

$$Prob'(\text{Path ends at terminal value } w'_t) = \frac{w'_t \times numPathsToExit[t]}{\sum_{j=1}^{K} w'_j \times numPathsToExit[j]}. \tag{6}$$

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 i^{th} 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—that is, $\langle j, i\rangle \in g.BReturnTuples[k] \Leftrightarrow (g.BReturnTuples[k])^{-1}[i] = j$.

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, for example, $numPathsToExit^{g.BConnections[k]}$) only considers matched paths from g's middle vertices 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, for example, $numPathsToExit^{g.AConnection}[k]$, 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 $g.BReturnTuples$):

$$Prob(m_{index}) = \frac{numPathsToExit^{\,g.A}[m_{index}] \times numPathsToExit^{\,g.B[m_{index}]}[(g.BRT[m_{index}])^{-1}[i]]}{g.numPathsToExit[i]}. \quad (7)$$

Example 6.3. Consider the CFLOBDD depicted in Figure 11, and suppose that the goal is to sample a matched path that leads to terminal value T. From Example 6.2, we know that (i) the outermost grouping has seven 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 Equation (7), we should sample the middle vertices with probabilities

$$Prob(m_{index} = 1) = \frac{[3,1][1]\times[3,1][2]}{7} = \frac{3\times1}{7} = \frac{3}{7} \qquad Prob(m_{index} = 2) = \frac{[3,1][2]\times[4][1]}{7} = \frac{1\times4}{7} = \frac{4}{7}.$$

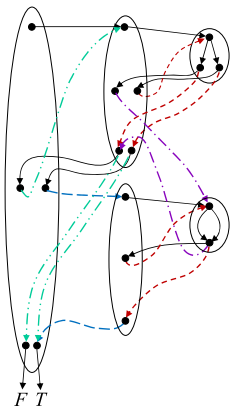


Fig. 11. CFLOBDD for $\lambda w, x, y, z.(w \wedge x) \vee (y \wedge z)$, with variable ordering $\langle w, x, y, z\rangle$.

Once m_{index} has been selected in accordance with Equation (7), 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" or "1" accordingly; for a *ForkGrouping*, the designated exit vertex—either 1 or 2—specifies a unique assignment: "0" or "1," respectively.

Pseudo-code for the assignment-sampling algorithm can be found in our previous work [59, Algorithm 22].

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 $O(\max(2^l, \text{size of argument CFLOBDD}))$.

7 CFLOBDD ALGORITHMS FOR MATRICES AND VECTORS

In this section, we discuss how to represent matrices and vectors using CFLOBDDs, and how to perform some important operations on them.

7.1 Representing Matrices and Vectors Using CFLOBDDs

Matrix Representation. We represent square matrices using CFLOBDDs by having the Boolean variables correspond to bit positions in the row and column indices. In other words, 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.[14] The indices of elements of matrices represented in this way

[14]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.

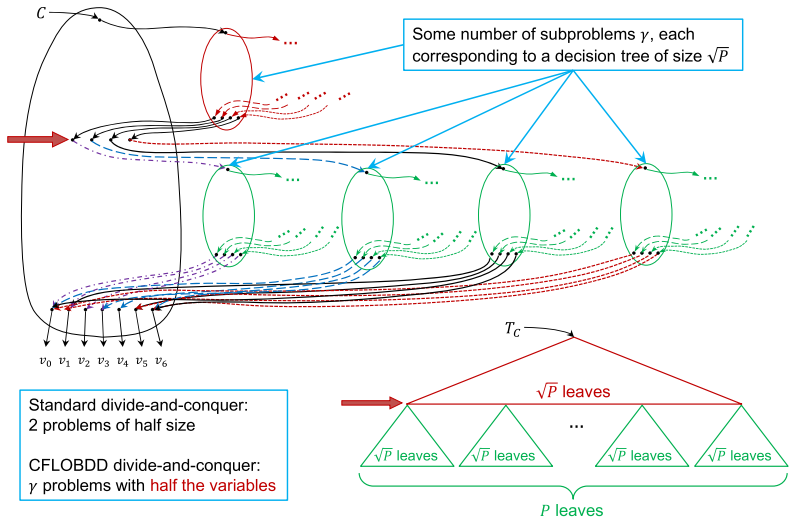


Fig. 12. Why a $\sqrt{P} \times \sqrt{P}$-decomposition is the natural problem decomposition for divide-and-conquer algorithms on structures represented as CFLOBDDs.

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.

There is an important, non-standard consequence of using a CFLOBDD to represent a matrix that very likely is not apparent from the preceding discussion, 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 Figure 12. 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 the tree partitioning shown in the lower-right corner of Figure 12: 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 de-

cision variable for the topmost ply is associated with the level-0 grouping found by following the A-connection of the A-connection of the ..., and so on. 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 (Section 7.3), 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 $\sqrt{P} = \sqrt{N^2} = N$ submatrices, 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: submatrices 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 two A-connection variables for each dimension of the block structure, and two B-connection variables for each dimension of the submatrix 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 $4 \times 4 = \sqrt{16} \times \sqrt{16}$. With level-4 CFLOBDDs, one has $n = 8$ variables for each dimension in the full-size matrix. Thus, there are four A-connection variables for each dimension of the block structure, and four B-connection variables for each dimension of the submatrix 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 previously 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 four submatrices, each of size 8×8, and decompose a matrix of size 256×256 iinto four submatrices, each of size 128×128).

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. 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 $2^n \times 1$.)

7.2 Kronecker Product

When using CFLOBDDs to represent matrices on which Kronecker products are performed, we typically use the interleaved-variable ordering. In the following, we describe two variants of the Kronecker product that result in different interleavings of the index variables of the argument matrices.

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—that is, $w_0v_0, w_0, v_1, \ldots, w_m v_{n-1}, w_m, v_n$. The grouping's A-connection is the proto-CFLOBDD of W, with return edges that map the i^{th} 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 j^{th} exit vertex to the exit vertex of the level $k+1$ grouping that corresponds to the value $w_i v_j$.
(3) If any of the values in the sequence $[w_i v_j : i \in [0..m], j \in [0..n]]$ 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.

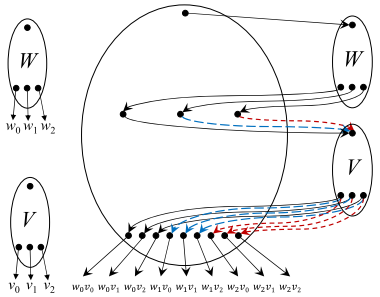


Fig. 13. Level-k CFLOBDDs for matrices W and V, and the level-$(k+1)$ CFLOBDD for $W \otimes V$.

The construction through step (2) is illustrated in Figure 13. Pseudo-code for the algorithm can be found in our previous work [59, 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).

Variant 2. There is a second way to perform a Kronecker product of W and V, which results in a representation of $W \otimes V$ that has the variable ordering $(x \bowtie w) \bowtie (y \bowtie z)$. Pseudo-code for that algorithm can be found in our previous work [59, Algorithm 17].

7.3 Matrix Multiplication

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 subproblem 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 submatrices 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 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 previously 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 7.1. 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:

$$\begin{bmatrix} ev_1 & ev_1 \\ ev_2 & ev_2 \end{bmatrix} \times \begin{bmatrix} ev'_1 & ev'_2 \\ ev'_1 & ev'_3 \end{bmatrix} = \begin{bmatrix} ev_1 ev'_1 + ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ ev_2 ev'_1 + ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix} = \begin{bmatrix} 2ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ 2ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix}. \quad (8)$$

Each entry in the right-hand-side matrix can be represented by a set of triples—that is,

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

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

$$[\{[(1,1),2]\},\{[(1,2),1],[(1,3),1]\},\{[(2,1),2]\},\{[(2,2),1],[(2,3),1]\}]. \quad (9)$$

Now suppose that the two matrices are submatrices of level-2 groupings connected by ReturnTuples $rt = [5,2]$ and $rt' = [6,1,2]$, respectively. Then applying $\langle rt, rt'\rangle$ to Equation (9) results in

$$[\{[(5,6),2]\},\{[(5,1),1],[(5,2),1]\},\{[(2,6),2]\},\{[(2,1),1],[(2,2),1]\}]. \quad (10)$$

We call the objects shown in Equations (9) and (10) *MatMultTuples*. By this device, the answer to a matrix-multiplication subproblem (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 Equation (8): 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 non-zero 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

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

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 Equation (9) to Equation (10). 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 Equation (9): $bp = \{[(1,2),1],[(1,3),1]\} = [(1,2) \mapsto 1, (1,3) \mapsto 1]$. Then the transformation of bp induced by rt and rt' can be expressed as follows (where Equation (11) expresses the general case):

$$\begin{aligned} \langle rt, rt'\rangle(bp) &\stackrel{\text{def}}{=} \{(rt(ev), rt'(ev')) \mapsto bp(ev, ev') \mid ev \in EV, ev' \in EV'\} \quad (11) \\ &= \{(rt(1), rt'(2)) \mapsto bp(1,2), (rt(1), rt'(3)) \mapsto bp(1,3)\} \\ &= \{(5,1) \mapsto 1, (5,2) \mapsto 1\}. \end{aligned}$$

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 *):

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

Pseudo-code for the matrix-multiplication algorithm can be found in our prior work [59, Algorithms 19 and 20].

7.4 Vector-to-Matrix Conversion

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 presented in Section 7.3.[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 $\log n + 1$.

Let $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ denote the variables in the CFLOBDD representation of V. The rows of M use the same x variables; the columns of M use a second set of n variables: $y = \langle y_0, y_1, \ldots, y_{n-1} \rangle$. M uses the interleaved ordering $x \bowtie y$.

Example 7.2. 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. The algorithm constructs 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. (Pseudo-code for the algorithm can be found in our prior work [59, Algorithm 18].)

8 RELATIONS EFFICIENTLY REPRESENTED BY CFLOBDDS

In this section, we prove that there exists an inherently exponential separation between CFLOBDDs and BDDs by showing that there is a family of functions f_n for which, for all $n = 2^l$, the CFLOBDD for f_n can be exponentially smaller than *any* BDD for f_n. Note that we do not assume any specific variable ordering when discussing the sizes of BDDs for the functions used to prove the separation. Moreover, our result applies to ROBDDs (in which "don't-care" nodes are removed and plies are skipped). As a proxy for memory, we use node counts in BDDs, and vertex counts and edge counts in CFLOBDDs. (Recall from footnote 4 that we use "node" solely for BDDs, whereas "groupings" and "vertices"—depicted as the dots inside groupings—refer to CFLOBDDs.)

We show this separation using the family of Hadamard relations, which represent the family $\mathcal{H}$ of Hadamard matrices discussed in Sections 2 and 3.4. The Hadamard matrices play a role in many quantum algorithms, including the seven that are used in Section 10.2.2 to evaluate the effectiveness of CFLOBDDs for simulating quantum circuits (namely, GHZ, BV, DJ, Simon's algorithm, **Quantum Fourier Transform (QFT)**, Shor's algorithm, and Grover's algorithm). See Section 9.2, Section 10.2.2, and our previous work [59, Section 9.3].

Theorem 8.1 (Exponential Separation for the Hadamard Relations). *The Hadamard relation $H_n : \{0,1\}^{n/2} \times \{0,1\}^{n/2} \to \{1,-1\}$ between variable sets $(x_0 \cdots x_{n/2})$ and $(y_0 \cdots y_{n/2})$, where*

[15] Matrix-vector multiplication is performed similarly.

$n = 2^l$, can be represented by a CFLOBDD with $O(\log n)$ vertices and edges. In contrast, a BDD that represents H_n requires $\Omega(n)$ nodes.

Proof. *CFLOBDD Claim.* As shown in Section 3, each matrix $H_n \in \mathcal{H}$, where $n = 2^l$ can be represented by a CFLOBDD with $O(l)$ vertices and edges—that is, with $O(\log n)$ 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. 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—that is, $\mathcal{T} \stackrel{\text{def}}{=} \forall k \in \{0..n/2-1\}, x_k \mapsto T, y_k \mapsto T$. 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. In other words, if $H_n[A_1] = v$ (where v is either 1 or -1), then $H_n[A_2] = -v$. We prove this claim by induction on level.

Proof. *Base Case*:

- $n = 2$. $H_2[A_1]$ is the lower-right corner of Figure 2(a), 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 (16^{th}) leaf in Figure 2(c), which yields a value of 1. For $k = 0$, A_2 ends up at the 12^{th} leaf, which is -1; if $k = 1$, A_2 ends up at the 15^{th} 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}—that is, $H_{2^m}[A_1] = v$ (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 all be mapped to the same value by the function represented by B, which violates the definition of the Hadamard relation H_n. (More precisely, $H_n[A_1] = 1$, for $n \geq 4$, but $H_n[A_2] = H_n[A_3] = -1$, for $n \geq 4$, which can be proved using an inductive argument similar to Case 1.) Consequently, (x_k, y_k) cannot be dropped as a pair.

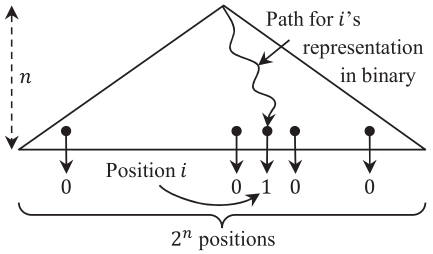


Fig. 14. One-hot encoding of the basis vector e_i as a decision tree. The single occurrence of 1 is at the leaf indexed by i—that is, at the end of the path from the root that follows the bits of i's representation in binary.

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, B—and hence any BDD that represents H_n—requires $\Omega(n)$ nodes. □

Unfortunately, we do not have 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). It could be that there are families of functions for which BDDs are exponentially more succinct than any corresponding CFLOBDD; however, it could also be that for every BDD there is a corresponding CFLOBDD no more than, say, a polynomial factor larger.

9 APPLICATIONS TO QUANTUM-CIRCUIT SIMULATION

For certain problems, algorithms run on quantum computers achieve polynomial to exponential speed-ups over their classical counterparts. In this section, we give background on quantum computing (Section 9.1), summarize the problems used in the experiments in Section 10.2.2 (Section 9.2), articulate some advantages of quantum-circuit simulation (Section 9.3), and discuss the potential of CFLOBDDs (Section 9.4).

9.1 Background on Quantum Computing

To make the article self-contained, we briefly summarize the quantum-computing model.

Qubits. In classical computing, a bit is usually thought of as having the value 0 or 1, and the state of a set of n bits $x_0, \ldots, x_{n-1}$ is a string in $\{0,1\}^n$. A different approach is based on 1-hot encodings [28]: a bit is either 0, encoded as $e_0 = \begin{smallmatrix}0\\1\end{smallmatrix}\begin{bmatrix}1\\0\end{bmatrix}$, or 1, $e_1 = \begin{smallmatrix}0\\1\end{smallmatrix}\begin{bmatrix}0\\1\end{bmatrix}$. The states of a two-bit state space would be encoded as follows: $e_{00} = \begin{smallmatrix}00\\01\\10\\11\end{smallmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $e_{01} = \begin{smallmatrix}00\\01\\10\\11\end{smallmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $e_{10} = \begin{smallmatrix}00\\01\\10\\11\end{smallmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, and $e_{11} = \begin{smallmatrix}00\\01\\10\\11\end{smallmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$. Quantum-computing generalizes the second approach: a *qubit* can have a value such as $\begin{smallmatrix}0\\1\end{smallmatrix}\begin{bmatrix}\alpha_0\\\alpha_1\end{bmatrix}$, where α_0 and α_1 are complex numbers, called *amplitudes*, such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$—in other words, a qubit is a complex unit vector in a vector space with basis vectors e_0 and e_1. For two qubits, the basis vectors are e_{00}, e_{01}, e_{10}, and e_{11}, and the state space consists of the complex unit vectors of the form $\begin{smallmatrix}00\\01\\10\\11\end{smallmatrix}\begin{bmatrix}\alpha_{00}\\\alpha_{01}\\\alpha_{10}\\\alpha_{11}\end{bmatrix}$, where $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$. In general, an n-qubit space consists of the complex unit vectors in a 2^n-dimensional space, and a quantum state can have non-zero amplitudes for all 2^n basis vectors. The decision tree for the one-hot encoding of a basis vector e_i, $0 \le i \le 2^n - 1$, of such a 2^n-dimensional space is depicted in Figure 14.

Quantum Circuits. 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 $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, and $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. 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);[16] X complements the indices of a qubit's basis states, and thus flips the positions of the amplitudes, sending $\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\begin{bmatrix} \alpha_0 \\ \alpha_1 \end{bmatrix}$ to $\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\begin{bmatrix} \alpha_1 \\ \alpha_0 \end{bmatrix}$. 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, for example, applies H to the j^{th} qubit of an n-qubit quantum state is $I^{\otimes(j-1)} \otimes H \otimes I^{\otimes(n-j)}$.

Measurement and Entanglement. A *measurement* operation samples a basis vector based on the distribution given by the squares of the absolute values of the amplitudes. Qubits are said to be *entangled* if the measurement of one qubit can influence the result obtained by measuring a different qubit. A **Controlled-NOT (CNOT)** gate operates on 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.' For two qubits, with the first qubit as the control-bit, the gate's matrix is $\begin{smallmatrix} & 00 & 01 & 10 & 11 \\ 00 & 1 & 0 & 0 & 0 \\ 01 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 0 & 1 \\ 11 & 0 & 0 & 1 & 0 \end{smallmatrix}$. For instance, $CNOT \times (\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}) = CNOT \times \begin{smallmatrix} 00 \\ 01 \\ 10 \\ 11 \end{smallmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{smallmatrix} 00 \\ 01 \\ 10 \\ 11 \end{smallmatrix}\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$. The latter state is entangled: measuring either (or both) qubits yields either the state e_{00} or e_{11}, each with probability $\frac{1}{2}$. In other words, after measuring either qubit and obtaining a value $v \in \{0, 1\}$, the value of the other qubit must also be v.

9.2 Quantum Algorithms

This section describes the quantum-computation problems that are used in the experiments in Section 10.2.2. More details about the algorithms for these problems can be found in our previous work [59, Section 9.3]

The GHZ State. The GHZ state is the following entangled state vector over three qubits (i.e., a unit vector of size 8): $GHZ_3 = \frac{1}{\sqrt{2}}[10000001]^T$. We extend the concept to n qubits by defining $GHZ_n = \frac{1}{\sqrt{2}}[100\ldots001]^T$, which is a unit vector of size 2^n.

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

The DJ Problem. Given an oracle that implements a function $f : \{0,1\}^n \to \{0,1\}$, 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.

Simon's Problem. 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 $x = y \oplus s$, find the hidden vector s.

Quantum Fourier Transform. The QFT is a linear transformation that, in matrix form, has the following form: $\begin{bmatrix} 1 & 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ldots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ldots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \ldots & \omega^{(N-1)(N-1)} \end{bmatrix}$, where $N = 2^n$ and $\omega = e^{2\pi i/N}$. The problem to be solved is to apply the QFT matrix to a given quantum-state vector.

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

Shor's Algorithm. The problem is to find prime factors of a given integer K, where $0 \leq K \leq 2^n - 1$.

Grover's Algorithm. The problem to be solved is the following search problem: given a function $f : \{0, 1, \ldots, 2^n - 1\} \rightarrow \{0, 1\}$ such that there is a unique x for which $f(x) = 1$, find x.

9.3 Advantages of Simulation

Simulation of a quantum circuit can have advantages compared to actually running a circuit on a quantum computer. 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. However, current quantum computers are error-prone, due to noise, and hence bad for confirming the correctness of a quantum algorithm. In contrast, a simulation never produces incorrect results (modulo floating-point round-off error and bugs in the implementation of the simulator).

Moreover, a simulation can deviate from certain requirements of the quantum-computation model and perform the simulation in a way that no quantum device could:

(1) 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* [72, 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*. This approach is particularly useful in simulating Grover's algorithm.

(2) 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. In some cases, our simulation of a quantum algorithm directly creates a CFLOBDD that represents an intermediate value, thereby avoiding a sequence of potentially more costly computational steps that stay within the quantum model. (See "A Special-Case Construction" in our previous work [59, Section 9.2.5], which is used in Grover's algorithm.)

(3) 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 simulation the quantum steps need only be performed once. At that point, there are two possibilities:
   - Once we have the final quantum state produced by the simulation in hand, one can inspect the amplitudes and thereby avoid doing any measurement at all.
   - If the goal of the simulation is to confirm that a given measurement protocol is correct, then because a simulated measurement does not cause any part of the simulated quantum state to be lost, multiple measurements can be made—for example, using the sampling algorithm given in Section 6.4.2—without having to re-perform the steps of the quantum computation before each successive measurement.

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: in a 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 previously. In principle, a more efficient simulation technique has the potential to change the threshold for quantum supremacy.

9.4 The Potential of CFLOBDDs for Quantum-Circuit Simulation

A quantum state $\sum_{w \in \{0,1\}^{2^n}} \alpha_w \cdot e_w$ is a function of type $\{0,1\}^n \to \mathbb{C}$ and thus could be encoded with a decision tree of height n. 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 following table indicates where to find details about the CFLOBDD operations needed to simulate a quantum circuit:

	This Article	Reference [59]	
State Construction			
Standard-basis vector	—	Section 9.2.4	Algorithm 25
Gate Construction			
Identity gate	—	Section 9.2.2	Algorithm 24
Hadamard gate	Figure 6(b)	Section 9.2.1	Algorithm 23
Not gate	—	Section 9.2.3	Variant of Algorithm 24
CNOT gate	—	Section 9.2.5	Appendix I
Operations			
Kronecker product	Section 7.2	Section 7.5	Algorithm 17
Matrix-matrix multiplication	Section 7.3	Section 7.7	Algorithm 19
Vector-matrix and matrix-vector multiplication	Sections 7.4 and 7.3	Section 7.6 + Section 7.7	Algorithms 18 and 19
Application of QFT	—	Section 9.2.6	Appendix J
Measurement	Section 6.4	Section 7.8	Algorithm 22

10 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)?

10.1 Experimental Setup

We compared our implementation of CFLOBDDs[17] against a widely used BDD package, CUDD [60] (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 four vCPUs, 16 GB of memory, and a stack size of 8,192 KB, running on Ubuntu OS. For RQ1 (Section 10.2.1), 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)** [18] (which can also be exponentially more succinct than BDDs), using Python package PySDD [41] (version 0.1). For RQ2 (Section 10.2.2), we used a set of quantum-simulation benchmarks, and again compared the performance of CFLOBDDs against CUDD. 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 that either directly or indirectly incorporate the answer sought. Our methodology is standard for quantum-simulation experiments. Each benchmark uses a

[17] The implementation is available at the following URL: https://github.com/trishullab/cflobdd

pre-processing step to create the CFLOBDD/BDD that represents the oracle. In each run, an answer is first generated randomly, 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 [24], a quantum-simulation library that uses tensor networks.

For the RQ2 experiments, we had to extend CUDD in two ways (for details, see our previous work [59, Section 10.1]):

(1) CUDD supports ADDs, which are MTBDDs with a value from a semiring at each terminal node. To support functions of type $\{0,1\}^n \rightarrow \mathbb{C}$, we needed a semiring that was not part of the standard CUDD distribution. We implemented a semiring of multi-precision-floating-point [20] complex numbers. 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).

10.2 Benchmarks and Experimental Results

10.2.1 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.

Table 2 shows the performance of CFLOBDDs, BDDs (both with and without dynamic variable-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 1 GB. 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 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 the second column of Table 2.

To understand how large a Boolean function could be created using CFLOBDDs (as a function of the number of Boolean variables),[18] we also measured the performance of the CFLOBDD

[18]The stack size was increased to 1 GB for the runs with more than $2^{2^{15}}$ Boolean variables.

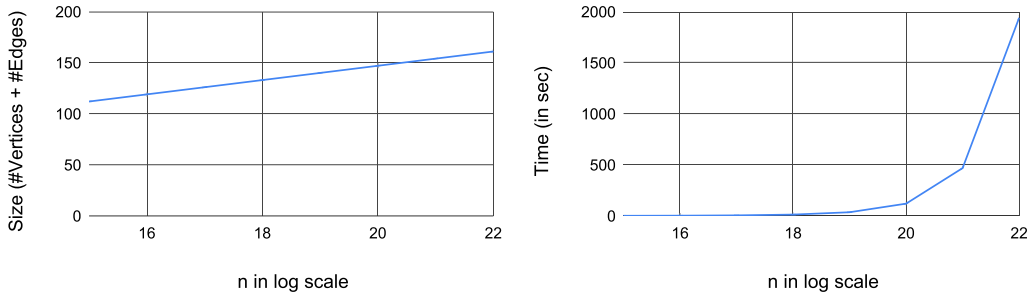
Table 2. Performance of CFLOBDDs against BDDs, BDDs with Dynamic Reordering, and SDDs on the Synthetic Benchmarks for Different Numbers of Boolean Variables

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 (15 min)				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 (15 min)					
	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 (15 min)		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)									

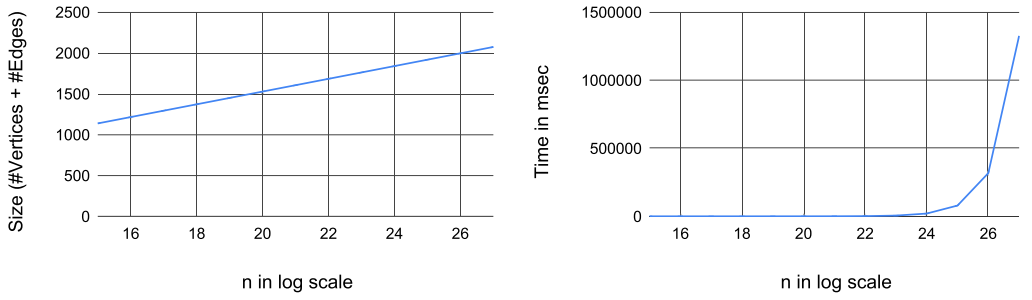
For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1 GB.

implementation on the micro-benchmarks using a timeout of 90 minutes. Figure 15 shows graphs of size (#vertices + #edges) and time versus the number of Boolean variables for the three benchmarks.[19] Figure 15(a) 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. Figure 15(b) and (c) 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.

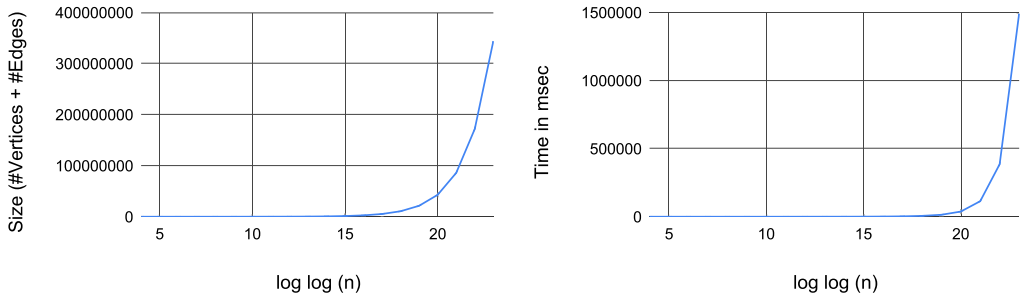
[19]Table 2 shows the comparison of CFLOBDDs, BDDs, and SDDs for examples with a 15-minute timeout. In contrast, Figure 15 shows the results of the stress test that we performed, where we gave the CFLOBDD implementation a 90-minute timeout.



(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)

Fig. 15. CFLOBDD performance with a timeout of 90 minutes. Note that in (c), the number of Boolean variables is on a log log scale.

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.

10.2.2 RQ2: Do CFLOBDDs Outperform BDDs When Used for Quantum Simulation (in Terms of Time and Space)? Tables 3 and 4 show the performance of CFLOBDDs and BDDs when simulating several well-known quantum algorithms. In each case, for both CFLOBDDs and BDDs, we used the interleaved-variable ordering.

Table 3. Performance of CFLOBDDs against BDDs for Increasing Numbers of Qubits

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)					

For GHZ, the algorithms do not depend on an input; the output is solely a function of the number of qubits used. We used the quantum circuit given by Yu and Palsberg [70, Section 2] for obtaining the GHZ state for n qubits; see also our previous work [59, Section 9.3.1]. For BV, DJ, QFT, Simon's algorithm, Shor's algorithm, and Grover's algorithm, we ran each algorithm

Table 4. Table (cont.) of the Performance of CFLOBDDs against BDDs for Increasing Numbers of Qubits

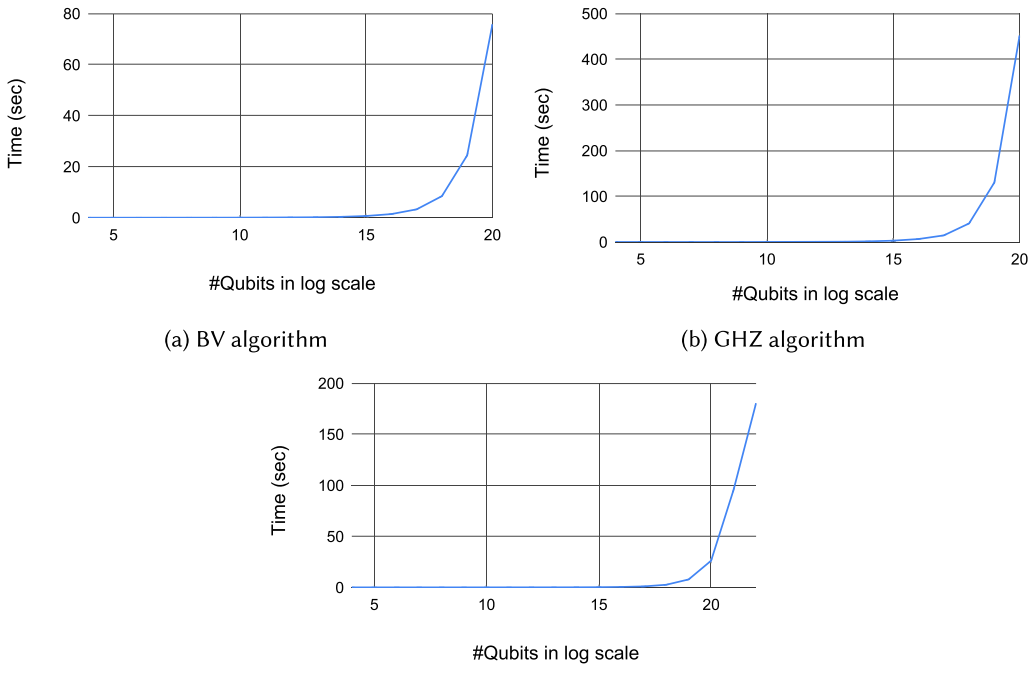
Benchmark	#Qubits	#Boolean Variables	CFLOBDD				BDD	
			#Vertices	#Edges	Total	Time (sec)	#Nodes	Time (sec)
Simon's Algorithm	16	64	583	16335	16918	0.71	**5512**	**0.275**
	32	128	123611	14096110	14219721	443.09	**80243**	**3.31**
	64	256	Timeout (90 min)				Timeout (90 min)	
QFT	4	8	7	73	80	0.001	**31**	**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 Algorithm $(N, a) = (15, 2)$	4	16	38	338	376	0.09	**69**	**0.04**
Shor's Algorithm $(N, a) = (21, 2)$	5	16	72	877	949	2.13	**136**	**0.72**
Shor's Algorithm $(N, a) = (39, 2)$	6	16	111	2443	2554	**12.6**	**187**	12.96
Shor's Algorithm $(N, a) = (69, 4)$	7	16	176	4331	4487	53.47	**605**	**30.38**
Shor's Algorithm $(N, a) = (95, 8)$	7	16	216	4928	5144	53.47	**974**	**41.47**
Shor's Algorithm $(N, a) = (119, 2)$	7	16	220	7533	7753	53.47	**3606**	**44.95**
Shor's Algorithm $(N, a) = (323, 2)$	9	32	Timeout (15 min)				Timeout (15 min)	
Grover's Algorithm	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)					

with 50 different randomly selected inputs, for each of the indicated number of qubits. Tables 3 and 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 Section 10.1); 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 Table 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."[20]

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. Figure 16 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.

[20]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

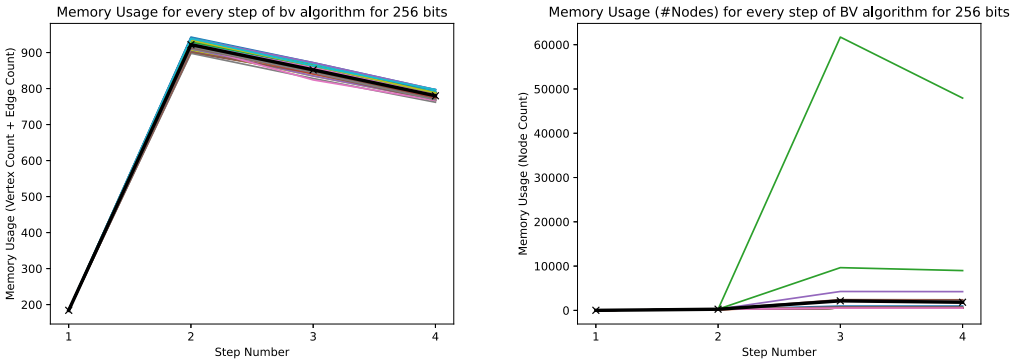
(c) DJ algorithm

Fig. 16. Execution time (in seconds) vs. number of qubits (on a log scale) for three of the benchmarks.

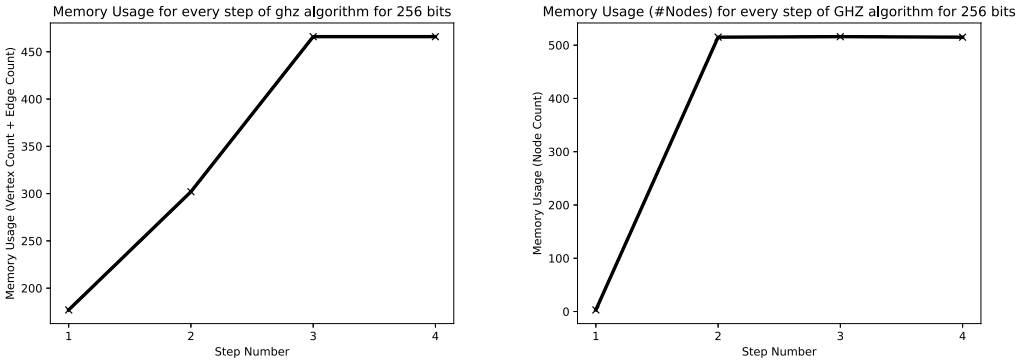
For both CFLOBDDs and BDDs, the transition from a problem size that completes successfully to a problem size that fails is rather abrupt. For all of the problems, the time reported for the CFLOBDD run with the largest number of qubits that completes successfully is well under 15 minutes. Unfortunately, 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.

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.

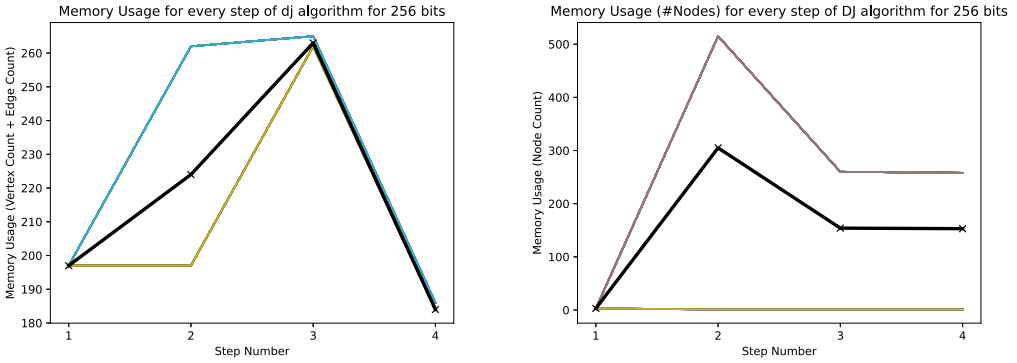
Intermediate Swell. For many of the algorithms, the initial and final CFLOBDD and BDD structures are of reasonable size, but there is an intermediate swell as the algorithm runs. Figures 17 and 18 show how size evolves in the various steps of five of the algorithms during the CFLOBDD-based and BDD-based simulations. The figures show how size evolves for all 50 runs, along with the average value at every step (highlighted in black). Figure 18 shows that the



(a) BV algorithm 256 qubits



(b) GHZ algorithm 256 qubits

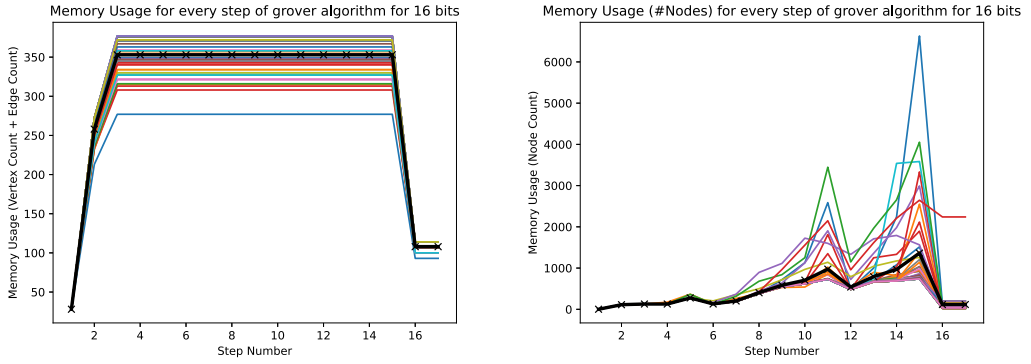


(c) DJ algorithm 256 qubits

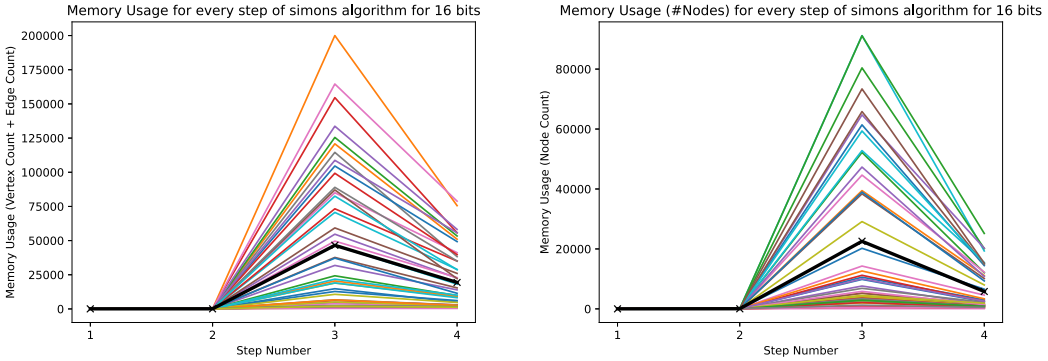
Fig. 17. Evolution of size during the indicated algorithms. (Left: CFLOBDDs; right: BDDs.)

CFLOBDD simulation of Grover's algorithm uses constant space from steps 3 through 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 [24], a state-of-the-art quantum simulator. Table 5 shows the performance of our CFLOBDD



(a) Grover's algorithm 16 qubits



(b) Simon's algorithm 16 qubits

Fig. 18. Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.)

implementation and Quimb on the previously discussed quantum benchmarks. For the Quimb-based simulations of GHZ, BV, DJ, and Grover's algorithm, we used MPSs (Matrix Product States) [7, 63] and MPOs (Matrix Product Operators) [62] in algorithms modeled after the ones described in the work of Woolfe [68]. 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 [49] and the nearest-neighbor circuit mentioned in the work of Fowler et al. [21]. 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 [8], but the internal gates are created directly, as mentioned in the work of Woolfe [68] (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.[21]

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 4,096-bit string s is chosen, then the Grover oracle ma-

[21] With 32 qubits, Quimb takes 1,496.6 sec ≈ 25 min.

Table 5. Performance of CFLOBDDs against Quimb on Quantum Benchmarks for Different Numbers of Qubits

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 Algorithm	16	**0.71**	2.56
	32	443.09	**17.34**
	64	Timeout (15 min)	**267**
	128		Timeout (15 min)
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 (15 min)
Shor's Algorithm $(N, a) = (15, 2)$	4	0.09	**0.08**
Shor's Algorithm $(N, a) = (21, 2)$	5	2.13	**0.1**
Shor's Algorithm $(N, a) = (39, 2)$	6	12.6	**0.11**
Shor's Algorithm $(N, a) = (69, 4)$	7	53.47	**0.12**
Shor's Algorithm $(N, a) = (95, 8)$	7	42.8	**0.12**
Shor's Algorithm $(N, a) = (119, 2)$	7	64.8	**0.12**
Shor's Algorithm $(N, a) = (323, 2)$	9	Timeout (15 min)	**0.27**
⋮	⋮		⋮
Shor's Algorithm $(N, a) = (6085, 8)$	12		**107.28**
Shor's Algorithm $(N, a) = (11611, 2)$	13		Out of Memory
Grover's Algorithm	16	**0.009**	3.26
	32	**0.012**	Timeout (15 min)
	⋮	⋮	
	4096	**909.86**	
	8192	Timeout (15 min)	

trix 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 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 (Section 6.1.2) and (ii) the identity matrix (see [59, Section 9.2.2 and Algorithm 24]). 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. 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 Table 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 Figure 18(a) for a plot of memory usage for the CFLOBDD implementation of Grover's algorithm for 16 qubits.)

11 RELATED WORK

CFLOBDDs were devised in the late 1990s; however, except for a rejected U.S. patent application posted on the USPTO site in 2002 [56], nothing about them has ever been published. Work on them was abandoned in 2002 due to not having found an application on which CFLOBDDs performed better than BDDs—other than some of the recursively defined spectral transforms, such as the Reed-Muller, inverse Reed-Muller, Hadamard, and Boolean Haar wavelet transforms [32]. In a 2009 blog post, Lipton [38] sketched a proposal for *Pushdown BDDs*, a BDD-like structure based on NPDAs. CFLOBDDs are closely related—they are based on DPDAs rather than NPDAs—which caused us to re-examine what relations CFLOBDDs could represent efficiently, and to discover that ADD_n was such a relation [59, Section 8.3]. This article combines the material from the patent application with our recent results on quantum simulation.

The idea behind CFLOBDDs was inspired by an obstacle encountered in interprocedural path profiling [44]. In generalizing the Ball and Larus [4] intraprocedural path-profiling scheme to allow profiling of path fragments that cross procedure boundaries, Melski and Reps found that an acyclic, non-recursive, interprocedural control-flow graph of size k could have 2^{2^k} matched paths. For instance, Melski [42] found that one 20,000-line program had $2^{400{,}000}$ such paths—which would require the instrumentation code to manipulate 400,000-bit numbers! From here, it was only a short distance to CFLOBDDs—how to interpret such graphs as representations of Boolean functions, how to implement the operations of a BDD-like API, how to maintain canonicity, and so on. The feature that threatened to sink the path-profiling scheme—double-exponential *explosion*—became the linchpin of a double-exponentially *compressed* representation of Boolean functions.

Over the years, many variants of BDDs have been proposed [57]. 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.

Examples of (acyclic) edge-weighted BDD variants include EVBDDs [36] and FEVBDDs [61]. 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.

Unweighted BDD variants include MTBDDs [14, 15], ADDs [3], FBDDs (Free BDDs) [64, Section 6], (BMDs (Binary Moment Diagrams) [12], HDDs (Hybrid Decision Diagrams) [13], Differential BDDs [2], and IBDDs (Indexed BDDs) [33]. Several of these BDD variants offers exponential compression over classical BDDs. However, because FBDDs, BMDs, HDDs, and IBDDs that encode, for example, the identity function need to examine each variable, the exponential-separation argument for CFLOBDDs from Section 8 carries over for all of these variants.

BDD variants that involve cyclic structures include **Linear Inductive Functions (LIFs)/Exponentially Inductive Functions (EIFs)** [26, 27] and **Cyclic BDDs (CBDDs)** [52].

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 Equation (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.

Other data structures that generalize BDDs are SDDs [18] and **Variable Shift SDDs (VS-SDDs)** [47]. 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. In CFLOBDDs, 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 greater ability to reuse substructures than SDDs and VS-SDDs. (Moreover, b can also be used as the A-connection grouping of g.) SDDs and VS-SDDs (and their quantitative generalizations, such as Probabilistic SDDs [34]) have not, so far, been used in matrix computations, and implementations of operations such as the 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 (Table 2). However, the relationship between these representations and CFLOBDDs merits future study.

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 error). Decision diagrams used for such simulation include QMDDs [46, 71] and TDDs [29]. Both of these are weighted BDD representations, and hence cannot be compared in an apples-to-apples way with CFLOBDDs, which are unweighted representations (i.e., the edges of a CFLOBDD do not have associated weights). However, to understand the potential of CFLOBDDs, we mention here how our experimental results with CFLOBDDs compare with the published data for QMDDs: CFLOBDDs perform better than the best published numbers on some algorithms (GHZ, BV, DJ, Grover) and worse on others (QFT, Shor) [72, Tab. 5.1].[22] 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 Table 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 article, variational quantum algorithms, which include a noise channel, can also be simulated using CFLOBDDs. Huang et al. [31] 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.

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 non-recursive procedures in programs can enable small programs to have many execution paths, and is the essence of the aforementioned observation by Melski and Reps 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 the Ph.D. thesis of Melski [43, Section 3.5.4]. Melski uses Yannakakis's notion of L-reachability [69] (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, for example, program entry to program exit [43, Section 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 [44, Section 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 [53, 55].) Melski observes, "the number of [unbalanced-left] paths through G^*_{fin} can be doubly exponential in the size of G^*_{fin}."[23]

These results are tantamount to the statement (proposed by one of the referees) that "there is a family of programs P_n, written with non-recursive procedures, that each would be exponentially

[22]Note that the number of qubits for Shor's algorithm reported in the work of Zulehner and Wille [72, Tab. 5.1] is the number of qubits of the circuit, whereas in Tables 4 and 5, #Qubits is the number of bits of the number N being factored, where #*Qubits*-of-circuit = 2 ∗ #bits-of-N.

[23]Unfortunately, the aforementioned work with the 20,000-line program that had $2^{400,000}$ paths (which is what prompted Melski and Reps [44] to realize that they were facing double-exponential explosion) was carried out after their CC'99 paper had been published. 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 . . ." [44, Section 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 [48, 67] and version-space algebras [25, 35, 51].)

larger if written without non-recursive procedures." In the 1970s, the literature on program schematology [50] 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 [39, 50]). Thus, it would have been natural for the schematology literature to contain a result of the form stated previously. 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 [17, 22].

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 $L(matched)$-paths is equal to the product of the numbers of $L(matched)$-paths through P and Q. Multiplicative amplification 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:

$$P(0) = 1 \qquad P(n+1) = P(n)^2.$$

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 [40]. 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

$$\mathcal{D}_0 = \{\text{true}, \text{false}\} \qquad \mathcal{D}_n = powerset(D_{n-1}).$$

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 $\Theta(n)$. In contrast, the size of the set that is represented by D_n is described by the following recurrence relation:

$$S(0) = 1 \qquad S(n+1) = 2^{S(n)},$$

whose solution is non-elementary: $S(n)$ is an exponential tower of 2s, $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (with n 2s), of height n.

12 CONCLUSION

This article described a new data structure—CFLOBDDs—for representing functions, matrices, relations, and other discrete structures. CFLOBDDs are a plug-compatible replacement for BDDs, and they can represent Boolean functions in a more compressed fashion than BDDs—exponentially smaller in the best case—and, again in the best case, double-exponentially smaller than the size of a Boolean function's decision tree. Moreover, we showed an inherently exponential separation between CFLOBDDs and ROBDDs: the CFLOBDD for a function g can be exponentially smaller than *any* ROBDD for g.

Our experiments compared the time and space usage of CFLOBDDs and BDDs on two types of benchmarks: (i) micro-benchmarks and (ii) quantum-simulation benchmarks. We found that the improvement in scalability with CFLOBDDs can be quite dramatic. These results support the conclusion that, for at least some applications, CFLOBDDs provide a much more compressed

representation of discrete structures than is possible with BDDs, thereby permitting much larger problem instances to be handled than heretofore.

APPENDICES

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 Algorithm 4, 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.

B PROOF OF THE LEXICOGRAPHIC-ORDER PROPOSITION

Proposition 4.1*[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 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—that is, 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 j^{th} 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 Figure 19(a)). In this case, the return edges $e_i \rightarrow i$ and $e_j \rightarrow j$ "cross." By structural invariant 2b of Definition 4.1, this situation 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.

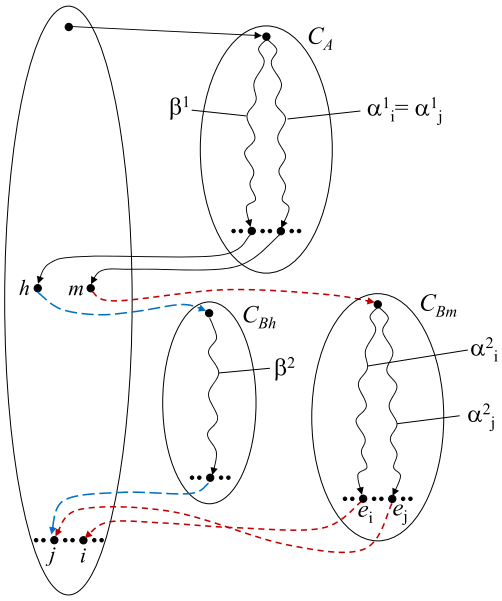
In this case, by the induction hypothesis applied to C_A, and the fact that $h < m$, it must be the case that we can choose β^1 so that $\beta^1 \prec \alpha_j^1$.

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.

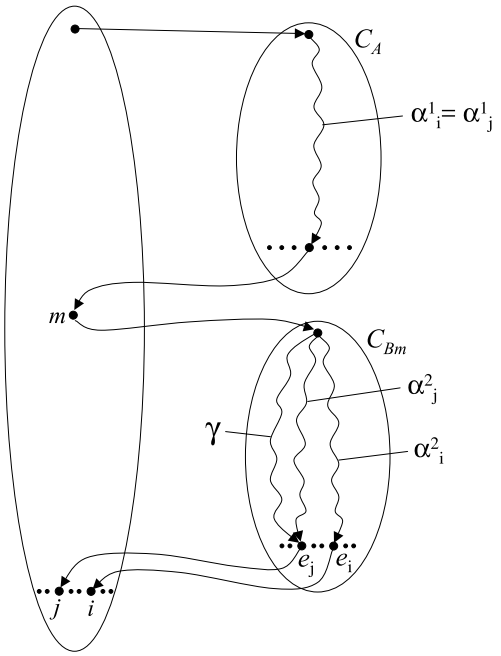
Case 1.B. Suppose that $e_j < e_i$ in C_{B_m} (see Figure 19(b)). Because $\alpha_i^2 \prec \alpha_j^2$, the induction hypothesis applied to C_{B_m} implies that there must exist an assignment $\gamma \prec \alpha_i^2 \prec \alpha_j^2$ that leads to e_j. In this case, we have that $\alpha_j^1 + \gamma \prec \alpha_j^1 + \alpha_j^2$, 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. $\alpha_i^1 \prec \alpha_j^1$.

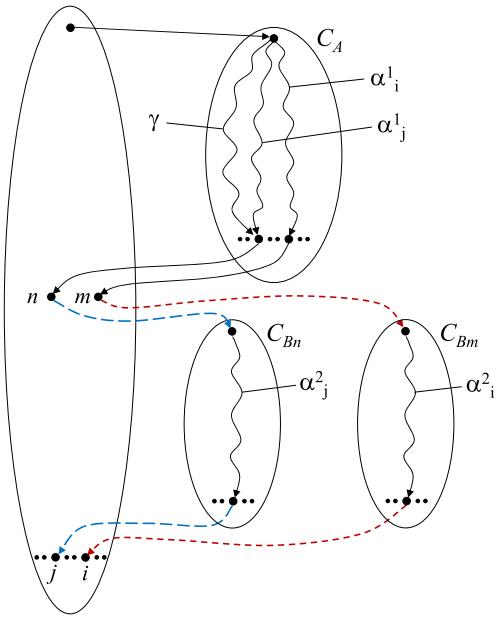
Because $\alpha_i^1 \prec \alpha_j^1$, the first halves of the matched paths followed during the interpretations of assignments α_i and α_j through C_A bring us to two different middle vertices of C, say m and n,



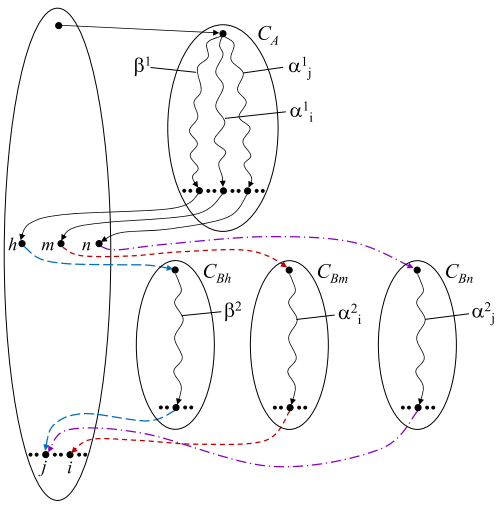
(a) Case 1.A of Proposition 4.1



(b) Case 1.B of Proposition 4.1.



(c) Case 2.A of Proposition 4.1.



(d) Case 2.B of Proposition 4.1.

Fig. 19. Cases 1.A, 1.B, 2.A, and 2.B of Proposition 4.1.

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 Figure 19(c).) The argument is similar to Case 1.B. By structural invariant 1 of Definition 4.1, $n < m$ 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 Figure 19(d).) The argument is similar to Case 1.A. By structural invariant 2 of Definition 4.1, 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 preceding cases, 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. □

C PROOF OF THE CANONICITY 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 through 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.

Figure 20(a) shows the decision tree for the function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. Figure 20(b) 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,[24] 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.

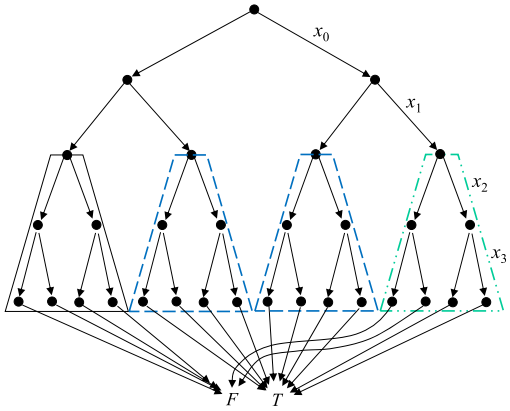
It is not hard to see that the structures created by Construction 1 obey the structural invariants that are required of CFLOBDDs by Definition 4.1:

- 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.
- 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).

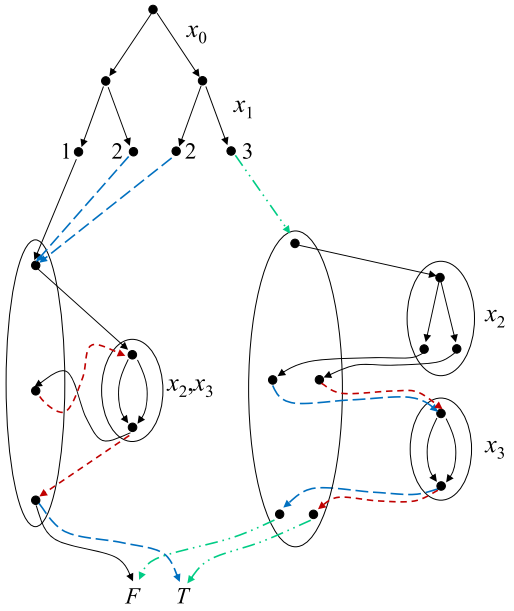
Moreover, Construction 1 preserves interpretation under assignments. Suppose that C_T is the level-k CFLOBDD constructed by Construction 1 for decision tree T; it is easy to show by induction on k that for every assignment α on the 2^k Boolean variables $x_0, \ldots, x_{2^k-1}$, the value obtained from 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

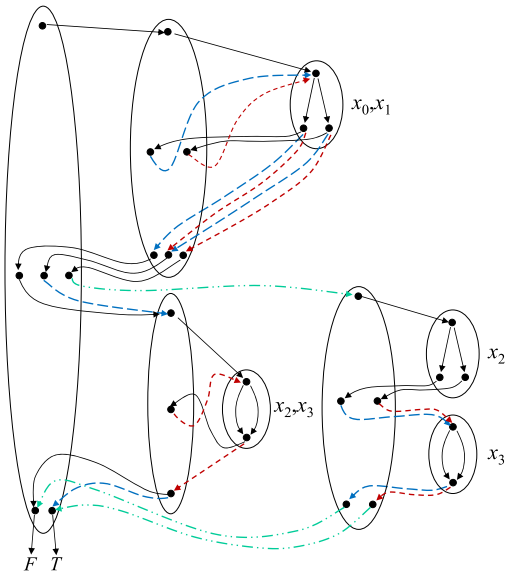
[24]The equality of the proto-CFLOBDDs is detected in step 5g.



(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. 5). For clarity, some of the level-0 groupings have been duplicated.

Fig. 20. Representations of the Boolean function $\lambda x_0 x_1 x_2 x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$.

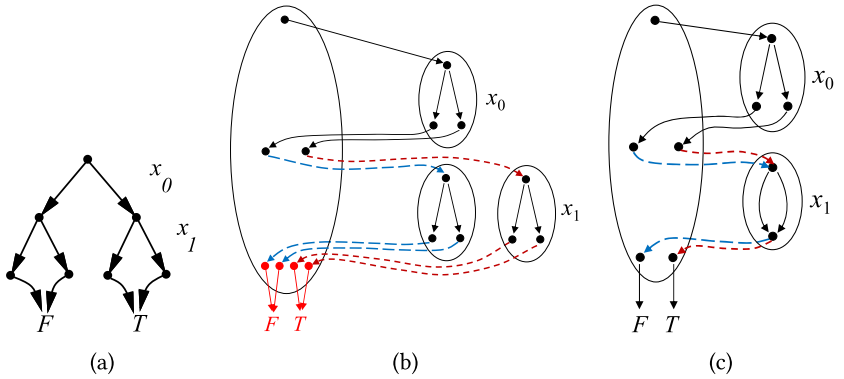


Fig. 21. (a) Decision tree for $\lambda x_0 x_1 . x_0$. (b) Fully expanded form of the CFLOBDD. (c) CFLOBDD.

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 Figure 20, *Unfold* would proceed from Figure 20(c), to Figure 20(b), 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 Figure 20(a).

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.
  – 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* order.)
  – At the end of step 3, a hybrid decision-tree/CFLOBDD object (à la Figure 20(b)) 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, Figure 22 shows the *Fold* trace generated by the application of Construction 1 to the decision tree shown in Figure 21(a) to create the CFLOBDD shown in Figure 21(c).
— An *Unfold trace* records the steps of *Unfold*(C):
  – CFLOBDD C is appended to the trace.

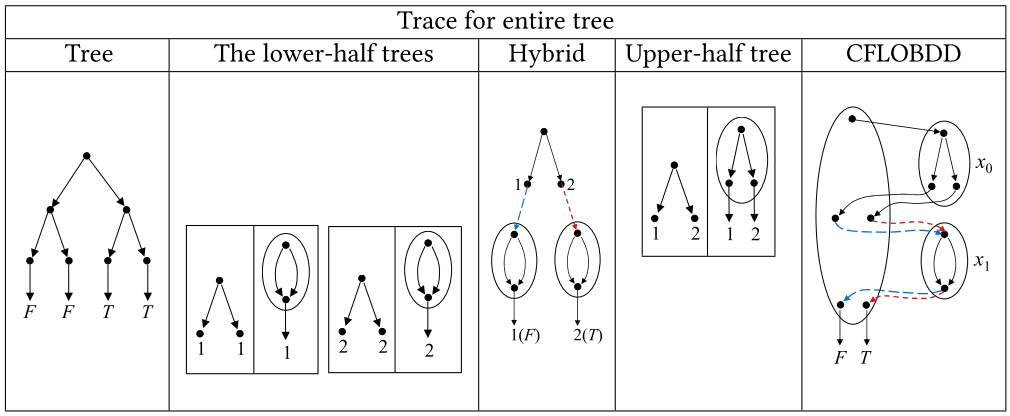


Fig. 22. The *Fold* trace generated by the application of Construction 1 to the decision tree shown in Figure 21(a) to create the CFLOBDD shown in Figure 21(c).

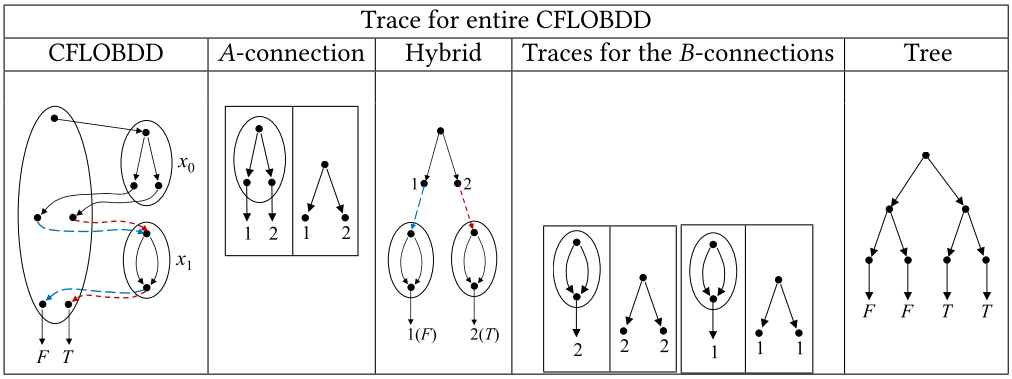


Fig. 23. The *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Figure 21(c) to create the decision tree shown in Figure 21(a).

- 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 Figure 20(b)) 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, Figure 23 shows the *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Figure 21(c) to create the decision tree shown in Figure 21(a).

Note how the *Unfold* trace shown in Figure 23 is the reversal of the *Fold* trace shown in Figure 22. We now argue that this property holds generally. (Technically, the argument given 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 (e.g., 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.[25] 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 which each leaf is labeled with the index of a middle vertex from the level-k+1 grouping of C (e.g., see Figure 20(b)).

As a consequence of Proposition 4.1, 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.[26] However, each middle vertex has an associated B-connection, and by structural invariants 2, 4, and 6 of Definition 4.1, 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).

(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 by Definition 4.1. 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'.

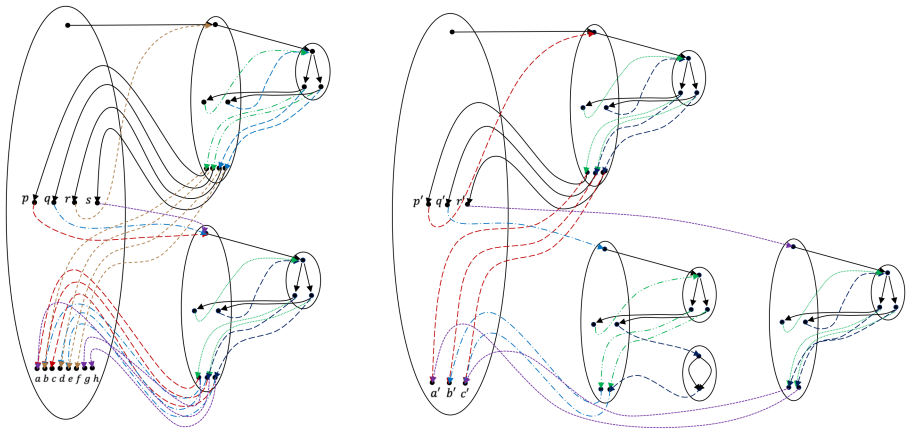
Consequently, FT is the reversal of UT, as was to be shown. □

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—that is, CFLOBDDs are a canonical representation of functions over Boolean arguments.

Theorem (4.3). *(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.*

[25] 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.

[26] This step is where the argument would break down if we attempt to apply the same argument to Figure 8(a). 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 Figure 8(a) is [1,2].



(a) Example C (b) Example C'

Fig. 24. $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$. The colors of the edges to proto-CFLOBDDs in C' correspond to the edges to the originating proto-CFLOBDDs in C.

D TIME COMPLEXITY OF REDUCE

In this section, we give a bound on the time complexity of the call on Reduce (Algorithm 10) in line 5 of BinaryApplyAndReduce (Algorithm 8). Let C be the level-l proto-CFLOBDD on which Reduce is invoked, and let 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.[27]

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.

Example D.1. Consider the level-2 proto-CFLOBDD C shown in Figure 24(a), 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 $P1 = [\{00\}, \{01\}, \{10\}, \{11\}]$—that is, 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

[27] 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 $|C|$.

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 $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$—that is, the vertices c, d, e, f, g, h are all mapped to exit vertex c. C' is shown in Figure 24(b). C' has only three exit vertices (a', b', c'). Consider how C is "reduced" 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 $\text{Reduce}(C_{B1}, [1, 2, 3])$ 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 in the following, 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 $\text{Reduce}(C_{B2}, [1, 2, 2])$. 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 $\text{Reduce}(C_{B3}, [1, 2, 2, 2])$. 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 $\text{Reduce}(C_{B2}, [1, 2, 2])$, 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 $\text{Reduce}(C_{B4}, [1, 2, 1])$—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 $\text{Reduce}(C_{B2}, [1, 2, 2])$ and $\text{Reduce}(C_{B3}, [1, 2, 2, 2])$ 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 $\text{Reduce}(C_A, [1, 2, 2, 3])$. 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 Example D.1, the cost-bound argument we give is slightly indirect. At a high level, it is structured as follows: we establish a relationship between $\text{Reduce}(C, \mathit{red})$ and that of a certain call on PairProduct (Theorem D.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 12). Theorem D.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 D.1. *Let C and C' be two proto-CFLOBDDs such that $C' = Reduce(C, red)$ for some reduction tuple red. Then $C = PairProduct(C, C')$.*[28]

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 (cf. [59, Section 6]). 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, *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

$$C' = \text{Reduce}(C, red), \tag{12}$$

$$\tilde{C} = \text{PairProduct}(C, C'). \tag{13}$$

C' created in Equation (12) 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 Equation (12) represents the same partition as C, and thus $\tilde{C}$ and C are equal by canonicity. □

In essence, Theorem D.1 shows that PairProduct(C, C') "undoes" all of the actions taken during Reduce(C, *red*).

Example D.2. Consider the result $\tilde{C}$ = PairProduct(C, C') for C and C' from Example D.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 from 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 $[e,c']$.

[28]To reduce clutter, we ignore the tuple of pairs of exit vertices that is returned by PairProduct (Algorithm 9), except for two places in Theorem D.3.

— 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 D.2. *(Local-Reduction Property). Let C and C' be two proto-CFLOBDDs such that $C' = Reduce(C, red)$ 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, $|g'| \leq |g|$. □

Example D.3. Consider the proto-CFLOBDDs C and C' from Figure 24. The size of the level-2 grouping g equals 1 (entry-vertex) + 4 (middle vertices) + (1 + 3) (1^{st} B-connection) + (1 + 3)(2^{nd} B-connection) + (1 + 4) (3^{rd} B-connection) + (1 + 3)(4^{th} B-connection) + 8 (exit vertices) = 30.

The size of g' equals 1 (entry-vertex) + 3 (middle vertices) + (1 + 3) (1^{st} B-connection) + (1 + 2)(2^{nd} B-connection) + (1 + 2) (3^{rd} 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 Theorem D.1 showed that PairProduct(C, C') "undoes" all of the actions taken during Reduce(C, *red*), Theorem D.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 D.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*	$C' = \text{Reduce}(C, red)$	$\text{PairProduct}(C, C')$
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, $\text{PairProduct}(C, C')$ returns a tuple that has C as the first component.

Moreover, the results produced by $\text{Reduce}(C)$ and $\text{PairProduct}(C, C')$ are of constant size, and could be implemented by table lookup. The return value from $\text{PairProduct}(C, C')$ is larger than the return value from $\text{Reduce}(C, red)$, 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 $C'_k = Reduce(C_k, red)$, for some reduction tuple *red*, $Cost(Reduce(C_k)) \leq Cost(PP(C_k, C'_k))$.

Consider two level-k+1 proto-CFLOBDDs, C'_{k+1} and C_{k+1}, such that $C'_{k+1} = \text{Reduce}(C_{k+1}, red)$. The proof breaks down into the following three cases:

(i) *A-connections*: PairProduct is first called recursively on $C_{k+1}.A$ and $C'_{k+1}.A$—that is, 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 $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, red_A)$ for some reduction tuple red_A. Thus, by the induction hypothesis,

$$Cost(Reduce(C_{k+1}.A)) \leq Cost(PP(C_{k+1}.A, C'_{k+1}.A)). \tag{14}$$

(ii) *B-connections*: The return value from the call on $\text{PairProduct}(C_{k+1}.A, C'_{k+1}.A)$ considered in the previous case is actually a tuple $[\tilde{C}_{k+1}.A, midVertexPairs]$. By the construction of C'_{k+1} from C_{k+1}, we know that $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, red_A)$ for some reduction tuple red_A, and thus by Theorem D.1, $\tilde{C}_{k+1}.A = C_{k+1}.A$.

For every $(i, j) \in midVertexPairs$, 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 $C'_{k+1}.B[j] = Reduce(C_{k+1}.B[i], red_{B[i]})$, for some $red_{B[i]}$.

We now consider the meaning of the pairs $(i, j) \in midVertexPairs$ from the standpoint of the language partitions used in Theorem D.1. Because C_{k+1} represents a finer partition of the strings in $\{0, 1\}^{2^{k+1}}$ than C'_{k+1}, the i^{th} exit vertex of $C_{k+1}.A$ represents a finer partition of the strings in $\{0, 1\}^{2^k}$ than the j^{th} 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 j^{th} 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 j^{th} middle vertex of C'_{k+1}. The reason this combining of languages took place during $\text{Reduce}(C_{k+1}, red)$ can only be because there were calls on $\text{Reduce}(C_{k+1}.B[i_1], red_1)$, $\text{Reduce}(C_{k+1}.B[i_2], red_2)$, …, $\text{Reduce}(C_{k+1}.B[i_p], red_p)$, for which the results were all equal to $C'_{k+1}.B[j]$. (The fifth bullet point of Example D.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,

$$\begin{aligned} Cost(Reduce(C_{k+1}.B[i_1])) &\le Cost(PP(C_{k+1}.B[i_1], C'_{k+1}.B[j])) \\ Cost(Reduce(C_{k+1}.B[i_2])) &\le Cost(PP(C_{k+1}.B[i_2], C'_{k+1}.B[j])) \\ &\cdots \\ Cost(Reduce(C_{k+1}.B[i_p])) &\le Cost(PP(C_{k+1}.B[i_p], C'_{k+1}.B[j])). \end{aligned} \tag{15}$$

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, $1 \le i \le e_A$ 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—that is, $j = \text{reductum}(i)$. We can now rewrite Equation (15) as

$$Cost(Reduce(C_{k+1}.B[i])) \le Cost(PP(C_{k+1}.B[i], C'_{k+1}.B[\text{reductum}(i)])). \tag{16}$$

(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 Lemma D.2, we know that $|g'| \le |g|$, and thus the cost of the call on `RepresentativeGrouping` in Reduce is no more than the cost of the call in PairProduct.

Finally, using Lemma D.2 and Equations (14) and (16), we obtain the desired result:

$$\begin{aligned} Cost(Reduce(C_{k+1})) &= |g'| + \sum_{i=1}^{e_A} Cost(Reduce(C_{k+1}.B[i])) + Cost(Reduce(C_{k+1}.A)) \\ &\le |g| + \sum_{i=1}^{e_A} Cost(Reduce(C_{k+1}.B[i])) + Cost(Reduce(C_{k+1}.A)) \\ &= |g| + \sum_{i=1}^{e_A} Cost(PP(C_{k+1}.B[i], C'_{k+1}.B[\text{reductum}(i)])) + Cost(PP(C_{k+1}.A, C'_{k+1}.A)) \\ &= Cost(PP(C_{k+1}, C'_{k+1})). \end{aligned}$$

□

ACKNOWLEDGMENTS

We thank Richard Lipton for the suggestion to apply CFLOBDDs to quantum simulation, Patrick Emonts for advice about the proper way to perform quantum-circuit simulation with tensor networks, and the referees for suggestions of how to clarify several items in the presentation. We are indebted to Alfons Laarman for helping us realize that the local-reduction property of Reduce (Lemma D.2) does not imply that a global-reduction property holds (footnote 27), and for suggesting that we look for a polynomial-time bound in terms of the sizes of Reduce's input and output proto-CFLOBDDs (Appendix D).

REFERENCES

[1] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. 2005. Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems* 27, 4 (2005), 786–818.

[2] Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. 1995. Differential BDDs. In *Computer Science Today: Recent Trends and Developments*. Lecture Notes in Computer Science, Vol. 1000. Springer, 218–233.
[3] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1993. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer Aided Design*. 188–191.
[4] Thomas Ball and James R. Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'96)*. 46–57.
[5] Thomas Ball and Sriram K. Rajamani. 2001. Automatically validating temporal safety properties of interfaces. In *Model Checking Software*. Lecture Notes in Computer Science, Vol. 2057. Springer, 103–122. https://doi.org/10.1007/3-540-45139-0_7
[6] Thomas Ball and Sriram K. Rajamani. 2001. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. ACM, 97–103. https://doi.org/10.1145/379605.379690
[7] Mari-Carmen Banuls, Matthew B. Hastings, Frank Verstraete, and J. Ignacio Cirac. 2009. Matrix product states for dynamical simulation of infinite chains. *Physical Review Letters* 102, 24 (2009), 240603.
[8] Stephane Beauregard. 2002. Circuit for Shor's algorithm using $2n+3$ Qubits. *arXiv preprint quant-ph/0205095* (2002).
[9] Michael Benedikt, Patrice Godefroid, and Thomas W. Reps. 2001. Model checking of unrestricted hierarchical state machines. In *Proceedings of the 28th International Colloquium on Automata, Language, and Programming (ICALP'01)*. 652–666. https://doi.org/10.1007/3-540-48224-5_54
[10] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. 1990. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. 40–45.
[11] Randal E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35, 6 (Aug. 1986), 677–691.
[12] Randal E. Bryant and Yirng-An Chen. 1995. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*. 535–541.
[13] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. 1995. Hybrid decision diagrams: Overcoming the limitations of MTBDDs and BMDs. In *Proceedings of the International Conference on Computer Aided Design*. 159–163.
[14] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. 1995. *Applications of Multi-Terminal Binary Decision Diagrams*. Technical Report CS-95-160. School of Computer Science, Carnegie Mellon University.
[15] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Chih-Yuan Yang. 1993. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*. 54–60.
[16] E. M. Clarke, M. Fujita, and X. Zhao. 1996. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of Discrete Functions*. T. Sasao and M. Fujita (Eds.). Kluwer Acadademic, Norwell, MA, 93–108.
[17] Robert L. Constable and David Gries. 1972. On classes of program schemata. *SIAM Journal on Computing* 1, 1 (1972), 66–118. https://doi.org/10.1137/0201006
[18] Adnan Darwiche. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*.
[19] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML*. ACM, 12–19. https://doi.org/10.1145/1159876.1159880
[20] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33, 2 (2007), 13–es.
[21] Austin G. Fowler, Simon J. Devitt, and Lloyd C. L. Hollenberg. 2004. Implementation of Shor's algorithm on a linear nearest neighbour qubit array. *arXiv preprint quant-ph/0402196* (2004).
[22] Stephen J. Garland and David C. Luckham. 1973. Program schemes, recursion schemes, and formal languages. *Journal of Computer and System Sciences* 7, 2 (1973), 119–160. https://doi.org/10.1016/S0022-0000(73)80040-6
[23] Eiichi Goto. 1974. *Monocopy and Associative Algorithms in Extended Lisp*. Technical Report TR 74-03. University of Tokyo, Tokyo, Japan.
[24] Johnnie Gray. 2018. quimb: A Python library for quantum information and many-body calculations. *Journal of Open Source Software* 3, 29 (2018), 819. https://doi.org/10.21105/joss.00819
[25] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010
[26] Aarti Gupta. 1994. *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction*. Ph.D. Dissertation. Carnegie Mellon University.
[27] Aarti Gupta and Allan L. Fisher. 1993. Representation and symbolic manipulation of linearly inductive Boolean functions. In *Proceedings of the International Conference on Computer Aided Design*. 192–199.

[28] David Harris and Sarah Harris. 2010. *Digital Design and Computer Architecture*. Morgan Kaufmann.
[29] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. 2020. A tensor network based decision diagram for representation of quantum circuits. *arXiv:2009.02618* (2020).
[30] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* 12, 1 (Jan. 1990), 26–60.
[31] Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. 2021. 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*. 456–472.
[32] Stanley Leonard Hurst, D. Michael Miller, and Jon C. Muzio. 1985. *Spectral Techniques in Digital Logic*. Academic Press.
[33] Jawahar Jain, James R. Bitner, Magdy S. Abadir, Jacob A. Abraham, and Donald S. Fussell. 1997. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Transactions on Computers* C-46, 11 (Nov. 1997), 1230–1245.
[34] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. 2014. Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on the Principles of Knowledge Representation and Reasoning*.
[35] James Koppel. 2021. Version space algebras are acyclic tree automata. *CoRR abs/2107.12568* (2021). https://arxiv.org/abs/2107.12568
[36] Yung-Te Lai and Sarma Sastry. 1992. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th Conference on Design Automation*. IEEE, 608–613.
[37] Ondrej Lhoták. 2006. *Program Analysis Using Binary Decision Diagrams*. Ph.D. Dissertation. McGill University.
[38] Richard J. Lipton. 2009. Gödel's Lost Letter and P=NP: BDD's and Factoring. Retrieved March 12, 2024 from https://rjlipton.wpcomstaging.com/2009/06/16/bdds-and-factoring
[39] Nancy A. Lynch and Edward K. Blum. 1979. A difference in expressive power between flowcharts and recursion schemes. *Mathematical Systems Theory* 12 (1979), 205–211. https://doi.org/10.1007/BF01776573
[40] Harry G. Mairson. 1992. A simple proof of a theorem of Statman. *Theoretical Computer Science* 103, 2 (1992), 387–394. https://doi.org/10.1016/0304-3975(92)90020-G
[41] Wannes Meert and Arthur Choi. 2018. PySDD, v0.1. Zenodo, 10.5281/zenodo.1202374. (March 2018). Retrieved March 12, 2024 from https://doi.org/10.5281/zenodo.1202374
[42] David Melski. 1998. Personal communication.
[43] David Melski. 2002. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation*. Ph.D. Dissertation. Computer Science Department, University of Wisconsin, Madison, WI.
[44] David Melski and Thomas Reps. 1999. Interprocedural path profiling. In *Compiler Construction*. Lecture Notes in Computer Science, Vol. 1575. Springer, 47–62.
[45] Donald Michie. 1967. *Memo Functions: A Language Feature with 'Rote-Learning' Properties*. Technical Report MIP-R-29. Department of Machine Intelligence and Perception, University of Edinburgh, Edinburgh, Scotland.
[46] D. Michael Miller and Mitchell A. Thornton. 2006. QMDD: A decision diagram structure for reversible and quantum circuits. In *Proceedings of the 36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. IEEE, 30–30.
[47] Kengo Nakamura, Shuhei Denzumi, and Masaaki Nishino. 2020. Variable shift SDD: A more succinct sentential decision diagram. *arXiv preprint arXiv:2004.02502* (2020).
[48] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock. 2021. Rewrite rule inference using equality saturation. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), Article 119, 28 pages.
[49] Michael A. Nielsen and Isaac L. Chuang. 2001. Quantum computation and quantum information. *Physics Today* 54, 2 (2001), 60.
[50] Michael S. Paterson and Carl E. Hewitt. 1970. Comparative schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, 119–127. https://dl.acm.org/doi/pdf/10.1145/1344551.1344563
[51] O. Polozov and S. Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.
[52] Frank Reffel. 1999. BDD-nodes can be more expressive. In *Proceedings of the Asian Computing Science Conference*.
[53] T. Reps. 1997. Program analysis via graph reachability. In *Proceedings of the International Logic Programming Symposium (ILPS'97)*. 5–19.
[54] T. Reps, S. Horwitz, and M. Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. 49–61.
[55] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. 49–61.

[56] Thomas W. Reps. 2002. Method for Representing Information in a Highly Compressed Fashion. (June 20, 2002). Retrieved March 12, 2024 from https://patentimages.storage.googleapis.com/ab/f4/17/2bbd2a0fad32f6/US20020078431A1.pdf
[57] Tsutomu Sasao and Masahira Fujita (Eds.). 1996. *Representations of Discrete Functions*. Kluwer Academic.
[58] Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice Hall.
[59] Meghana Sistla, Swarat Chaudhuri, and Thomas W. Reps. 2022. CFLOBDDs: Context-free-language ordered binary decision diagrams. *CoRR abs/2211.06818* (2022). https://doi.org/10.48550/arXiv.2211.06818
[60] Fabio Somenzi. 2012. *CUDD: CU Decision Diagram Package–Release 2.4.0.* University of Colorado at Boulder.
[61] Paul Tafertshofer and Massoud Pedram. 1997. Factored edge-valued binary decision diagrams. *Formal Methods in System Design* 10, 2 (1997), 243–270.
[62] Frank Verstraete, Juan J. Garcia-Ripoll, and Juan Ignacio Cirac. 2004. Matrix product density operators: Simulation of finite-temperature and dissipative systems. *Physical Review Letters* 93, 20 (2004), 207204.
[63] Guifré Vidal. 2003. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters* 91, 14 (2003), 147902.
[64] Ingo Wegener. 2000. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics.
[65] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*.
[66] J. Whaley and M. Lam. 2004. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. 131–144.
[67] M. Willsey. 2021. Fast and Extensible Equality Saturation with Egg. Retrieved March 12, 2024 from https://blog.sigplan.org/2021/04/06/equality-saturation-with-egg/
[68] Kieran Woolfe. 2015. *Matrix Product Operator Simulations of Quantum Algorithms*. Ph.D. Dissertation. School of Physics, University of Melbourne.
[69] M. Yannakakis. 1990. Graph-theoretic methods in database theory. In *Proceedings of the Symposium on Principles of Database Systems*. 230–242.
[70] Nengkun Yu and Jens Palsberg. 2021. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 542–558.
[71] Alwin Zulehner and Robert Wille. 2019. Advanced simulation of quantum computations. *IEEE Transactions on Computer Aided Design of Integrated Circuits and System* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427
[72] Alwin Zulehner and Robert Wille. 2020. *Introducing Design Automation for Quantum Computing*. Springer.

Received 12 May 2023; revised 7 December 2023; accepted 23 February 2024