

CFLOBDDs:

Context-Free-Language Ordered Binary Decision Diagrams

MEGHANA APARNA SISTLA, The University of Texas at Austin, USA

SWARAT CHAUDHURI, The University of Texas at Austin, USA

THOMAS REPS, University of Wisconsin-Madison, USA

This paper 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 useful for representing certain classes of functions, matrices, graphs, relations, etc. 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 GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover’s Algorithm, besting BDDs by factors of 128 \times , 1,024 \times , 8,192 \times , and 128 \times , respectively.

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

1 INTRODUCTION

Many areas of computer science—such as hardware and software verification, logic synthesis, and equivalence checking of combinatorial circuits—require a space-efficient representation of data, as well as space- and time-efficient operations on data stored in such a representation. Many of the tasks in the aforementioned areas involve operations on either (i) Boolean functions, or (ii) non-Boolean-valued functions over Boolean arguments. In some cases, a level of encoding is involved: the data of interest could be decision trees, graphs, relations, matrices, circuits, signals, etc., which are encoded as functions of type (i) or (ii). Binary Decision Diagrams (BDDs) [17] 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 an ROBDD (Reduced Ordered BDD) data structure. All manipulations of these Boolean functions are carried out using algorithms that operate on ROBDDs. ROBDDs are BDDs in which the same variable ordering is imposed on the Boolean variables (“Ordered”), and so-called *don’t-care* nodes are removed (“Reduced”). ROBDDs with non-binary-valued terminals are called Multi-Terminal BDDs (MTBDDs) [19, 21] or Algebraic Decision Diagrams (ADDs) [7]. We will refer to ROBDDs/MTBDDs/ADDs generically as BDDs from hereon.

Authors’ addresses: Meghana Aparna Sistla, The University of Texas at Austin, USA, mesistla@utexas.edu; Swarat Chaudhuri, The University of Texas at Austin, USA, swarat@cs.utexas.edu; Thomas Reps, University of Wisconsin-Madison, USA, reps@cs.wisc.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.

XXXX-XXXX/2024/5-ART \$15.00

<https://doi.org/10.1145/nnnnnnnnnnnnnn>

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) [9]. 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 [64] to handle relations over valuations over a Boolean program’s Boolean variables [10].
- The Datalog solver bddbddb, which uses BDDs as the backing representation of relations, was developed by Whaley and Lam to support a variety of program analyses [76, 77].
- Lhoták used BDDs in interprocedural program analyses to represent and manipulate collections of large sets, allowing him to use larger programs than previous studies of the factors that affect analysis precision [45].

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

Similar to BDDs, CFLOBDDs can represent functions, matrices, graphs, relations, etc. (using either binary-valued or multi-valued terminals, as appropriate), again with the possible advantage of an exponential degree of compression over BDDs. Even for objects that do not fall into the best-case scenario of perfect double-exponential compression, CFLOBDDs may provide better compression than BDDs. Like BDDs, CFLOBDDs are *canonical* (§4), and operations are performed on them directly (§7): they are never unfolded to the full decision tree. Moreover, an implementation can ensure that only a single representative is ever constructed for a given function; consequently, the test of whether two CFLOBDDs represent equal functions can be performed merely by comparing the values of two pointers.

CFLOBDDs are based on the following key insight:

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

The advantages of this idea are two-fold. First, whereas a BDD of size n can have at most 2^n paths, the “procedure-call” mechanism in CFLOBDDs allows a CFLOBDD of size n to have 2^{2^n} paths (§3.5.1). This difference is what lies behind the potential compression advantage of CFLOBDDs. Second, even when best-case compression is not possible, such “procedure calls” allow there to be additional sharing of structure beyond what is possible in BDDs: a BDD can share sub-DAGs, whereas a procedure call in a CFLOBDD shares the “middle of a DAG.” (See Figs. 3 and 6.)

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 (§10.2.1) and for quantum simulation (§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 (§3). In the best case, a CFLOBDD obtains *double-exponential compression in space*: i.e., 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 (§7). 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 (§8).
- We give efficient CFLOBDD representations for matrices used in quantum algorithms (§9).
- We measured the performance of CFLOBDDs and BDDs on synthetic and quantum-simulation benchmarks (§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 GHZ; 1,024× for BV; 8,192× for DJ; and 128× for Grover’s algorithm.

Organization. §2 reviews decision trees and BDDs. §3 introduces the basic principles underlying CFLOBDDs. §4 introduces some additional structural invariants that allow us to establish that each Boolean function has a unique, canonical representation as a CFLOBDD. §5 discusses how some standard techniques—hash-consing [31], function-caching (or *memo functions* [54]), and reference counting—apply to CFLOBDDs. §6 gives a denotational definition of the function that a CFLOBDD represents. §7 presents algorithms for a variety of CFLOBDD operations. §8 demonstrates an exponential gap between CFLOBDDs and BDDs: the CFLOBDD for a function f can be exponentially smaller than any BDD for f . §9 turns to the application of CFLOBDDs for quantum simulation, and shows how CFLOBDDs can represent efficiently some special matrices used in quantum algorithms. §10 poses two experimental questions and presents the results of experiments on synthetic and quantum-simulation benchmarks. §11 discusses related work. §12 concludes. Additional details are provided in Appendices §A–§L.

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

The theme of this paper is compressibility of Boolean functions, for which we need a family of examples that are indexed by some parameter. This section presents the set of *Hadamard matrices* \mathcal{H} , which recur in §3 and §8–§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 Fig. 1 as the base case. where \otimes denotes

$$H_2 = \begin{bmatrix} 0 & 1 \\ x_0 & 0 \\ 1 & -1 \end{bmatrix} \quad H_4 = H_2 \otimes H_2 = \begin{bmatrix} 0 & 1 \\ x_0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} H_2 & H_2 \\ H_2 & -H_2 \end{bmatrix} = \begin{bmatrix} 00 & 01 & 10 & 11 \\ x_0x_1 & 10 & 11 & 11 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

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

Kronecker product.¹ Fig. 1 shows H_2 and H_4 , the first two matrices in \mathcal{H} . The *Kronecker product* of two matrices is defined as

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

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

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

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

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

Decision Trees. A *decision tree* is a tree representation of a Boolean function. For a Boolean function B in $\{F, T\}^n \rightarrow W$, the decision tree T_B for B is a 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 $[v_1 \mapsto b_1, \dots, v_n \mapsto b_n]$ (where $b_1, \dots, b_n \in \{F, T\}$) 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 $\langle v_1, \dots, v_n \rangle$.)

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^{2^i} 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 $H_{2^{2i-1}}$. Not only do we avoid having to write a doubly superscripted subscript, we will see in §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.

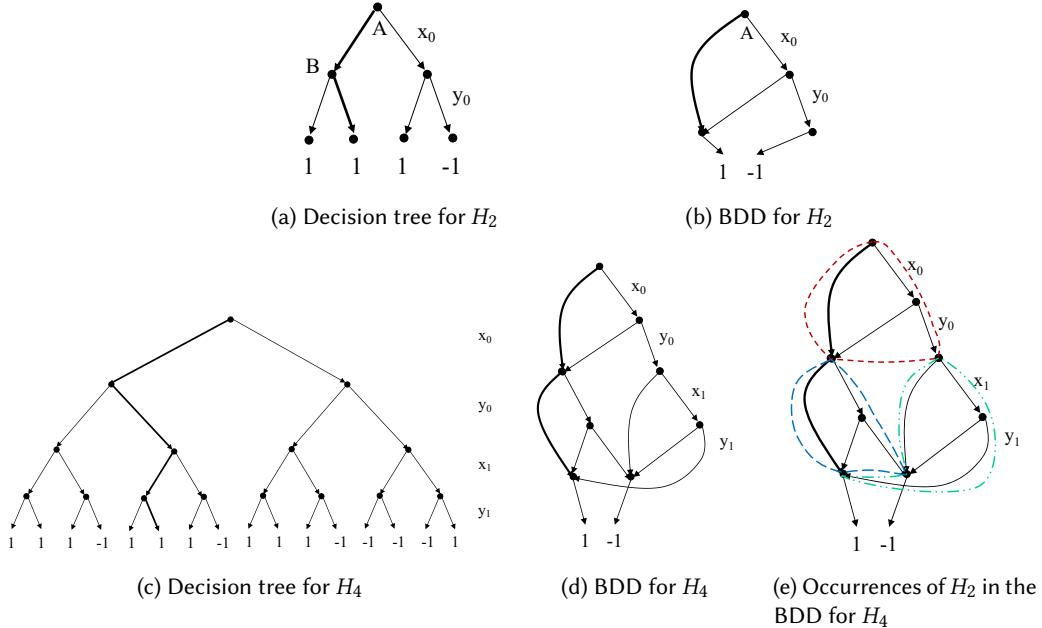


Fig. 2. Decision trees and BDDs for H_2 and H_4 , with plies in interleaved most-significant-bit order— $\langle x_0, y_0 \rangle$ and $\langle x_0, y_0, x_1, y_1 \rangle$, respectively. The bold paths show the assignments $[x_0 \mapsto F, y_0 \mapsto T]$ (for $H_2[0, 1]$) and $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$ (for $H_4[0, 3]$), respectively.

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

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

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

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

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

BDDs. A BDD is a compressed representation of a decision tree. Fig. 2b shows the BDD for H_2 , using the variable ordering $\langle x_0, y_0 \rangle$. Again, left branches are for F (or 0); right branches are for T (or 1). In the H_2 matrix, rows 0 and 1 are different, and hence the BDD node for x_0 is a *fork_node*, which forks to two different substructures. In row 0 of the matrix, columns 0 and 1 are identical, and hence the y_0 ply is skipped in the F branch of x_0 , with the F branch of x_0 leading directly to the terminal value 1. Conversely, in row 1 of the matrix, the columns differ, and hence the BDD node for y_0 in the T branch of x_0 is a fork_node. In Fig. 2b, the bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0, 1]$. (Only the edge for $x_0 \mapsto F$ is highlighted because the ply for y_0 is skipped when $x_0 \mapsto F$.)

Fig. 2d shows the BDD for H_4 under the interleaved-variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0, 3]$. (The path in the BDD only shows $x_0 \mapsto F, x_1 \mapsto F$ because the plies for y_0 when $x_0 \mapsto F$, and y_1 when $x_0 \mapsto F$ and $x_1 \mapsto F$ are skipped.)

Fig. 2e shows that the Kronecker product $H_4 = H_2 \otimes H_2$ corresponds to *stacking BDDs*—in essence, each terminal of the BDD for the left occurrence of H_2 in “ $H_2 \otimes H_2$ ” is replaced by a copy of H_2 . The BDD for H_4 contains three occurrences of H_2 : one in the $\langle x_0, y_0 \rangle$ plies, and two in the $\langle x_1, y_1 \rangle$ plies. The leftmost $\langle x_1, y_1 \rangle$ occurrence (blue-dashed outline) accounts for the three occurrences of matrix H_2 in the H_4 matrix; the rightmost occurrence (green dashed-double-dotted outline) corresponds to the negated matrix $-H_2$ in the lower-right corner of H_4 (cf. Fig. 1). Consequently, one can construct a BDD for each member of \mathcal{H} by repeated stacking, doubling the number of plies each time, per $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$, but only *tripling* the size with each such stacking operation (e.g., $H_8 = H_4 \otimes H_4$ has three copies of H_4 , etc.). Consequently, the size of the BDD for H_{2^i} is $O(3^i)$.

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

In contrast, CFLOBDDs employ a different principle than stacking to account for Kronecker product. Looking ahead, this principle is explained in §3.4, and as we will see when we get to Fig. 6b, there is a CFLOBDD of size $O(i)$ that encodes H_{2^i} . Consequently, CFLOBDDs achieve *double-exponential* compression on \mathcal{H} . Moreover, in §8.2, we show that this exponential separation is inherent: a BDD that represents H_{2^i} requires $\Omega(2^i)$ nodes (Thm. 8.3).

In the remainder of the paper, 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 paper gives what is essentially a complete account of CFLOBDD operations and invariants, and we 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 paper. For background about how BDDs are implemented, the reader is referred to Brace et al. [16].

3 CFLOBDDS

CFLOBDDs are a binary decision diagram inspired by BDDs, but the two data structures are based on different principles. A BDD is an acyclic finite-state machine (modulo ply-skipping), whereas a CFLOBDD is a particular kind of *single-entry, multi-exit, non-recursive, hierarchical finite-state*

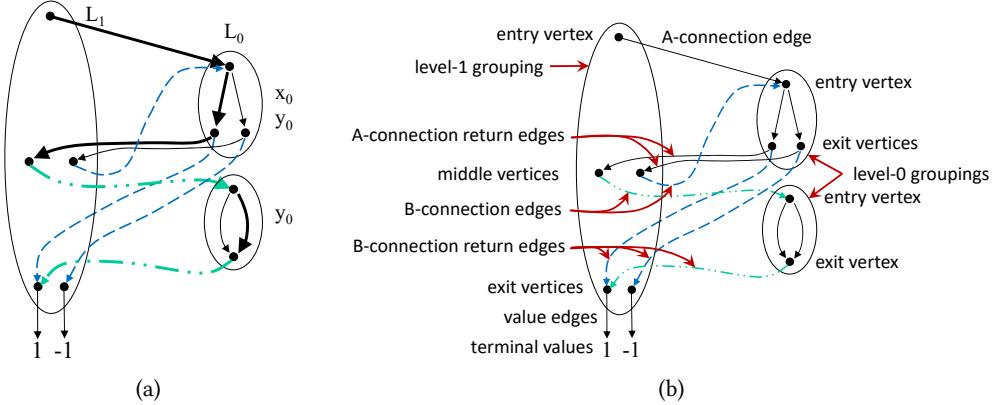


Fig. 3. (a) CFLOBDD for H_2 using the variable ordering $\langle x_0, y_0 \rangle$. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$ for $H_2[0, 1]$. (b) Guide to the terminology introduced in Defn. 3.1.

machine (HFSM) [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 $f : \{0, 1\}^n \rightarrow [1 \dots m]$ over variables x_0, \dots, 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, \dots, x_{n-1} . Functions g_0 and g_1 can be combined to yield f by the identity $f = \bar{x}_0 \cdot g_0 + x_0 \cdot g_1$ (where \bar{x}_0 denotes the complement of x_0 , “.” denotes logical-and, and “+” denotes logical-or). (See [22, §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: $x_0, \dots, x_{n/2-1}$ and $x_{n/2}, \dots, x_{n-1}$.² Let g_0 be the function of the first $n/2$ variables that maps them to $[1 \dots k]$, 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 $i \in [1 \dots k]$, 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 the functions g_0, g_1, \dots, g_k . Moreover, the divide-the-variables-in-half decomposition is carried out recursively on g_0, g_1, \dots, g_k , with mutual sharing of the decomposed functions that arise at all levels.

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

²For 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 §4.2 of property (2).)

³We are being deliberately vague about how g_0, g_1, \dots, g_k are combined, because the details are somewhat complicated. See Defns. 3.1 and 4.1 for the precise definition.

3.1 Matched Paths

The CFLOBDD representation of H_2 consists of three *groupings*, shown as three ovals in Fig. 3a.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 Fig. 3a is called a *fork grouping*.
- A level-0 grouping like the one at the lower right in Fig. 3a 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 Fig. 3a that leads to either 1 or -1 . Each such path starts from the entry vertex of the level-1 grouping, making "decisions" for the next variable in sequence each time the entry vertex of a level-0 grouping is encountered.

Fig. 3a illustrates the key principle behind CFLOBDDs—namely, the use of a *matching condition on paths*. The bold path is for the assignment $[x_0 \mapsto F, y_0 \mapsto T]$, which corresponds to $H_2[0, 1]$. The path starts at the level-1 grouping's entry vertex and goes to the entry vertex of the level-0 fork grouping via a solid edge (—); takes the left branch of the fork grouping (corresponding to $x_0 \mapsto F$); and leaves the fork grouping via a solid edge (—), reaching the leftmost of the middle vertices of the level-1 grouping. The path then goes to the entry vertex of the level-0 don't-care grouping via a dashed-double-dotted edge (— · · —); takes the right branch of the don't-care grouping (corresponding to $y_0 \mapsto T$); and leaves via a dashed-double-dotted edge (— · · —), reaching the leftmost exit vertex of the level-1 grouping, which is connected to the terminal value 1 (the value of $H_2[0, 1]$). A pair of incoming and outgoing edges of a grouping, such as the pairs of black solid edges and green dashed-double-dotted edges in the bold path in Fig. 3a, 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 [80]. (This idea is the origin of "CFL" in "CFLOBDD".) One way to formalize the condition is to label each edge from level i to level $i - 1$ with an open-parenthesis symbol of the form " $(_b$ ", where b is an index that distinguishes the edge from all other edges to any entry vertex of any grouping of the CFLOBDD. (In particular, suppose that there are NumConnections such edges, and that the value of b runs from 1 to NumConnections.) Each return edge that runs from an exit vertex of the level $i - 1$ grouping back to level i , and corresponds to the edge labeled " $(_b$ ", is labeled " $)_b$ ". Each path in a CFLOBDD then generates a string of parenthesis symbols formed by concatenating, in order, the labels of the edges on the path. (Unlabeled edges in the level-0 groupings are ignored in forming this string.) A path in a CFLOBDD is called a *matched-path* iff the path's word is in the language $L(\text{matched})$ of balanced-parenthesis strings generated by

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

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."

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

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

The matched-path principle allows a given grouping to play multiple roles during the evaluation of a Boolean function. In particular, the level-0 groupings are shared, and thus are used to interpret *different variables at different places in a matched path through a CFLOBDD*. For example, the level-0 fork grouping in Fig. 3a 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 Fig. 3a, the fork grouping is labeled with x_0 and y_0 , and the don’t-care grouping with y_0 . However, because the level-0 groupings interpret different variables at different places in a matched path, in later diagrams the level-0 groupings are generally not labeled with specific variables. In general, the principle is as follows:

Contextual-Interpretation Principle. *A level-0 grouping is not associated with a specific Boolean variable. Instead, the variable that a level-0 grouping refers to is determined by context: the 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 Fig. 3a contains cycles. That is, if one ignores the ovals in Fig. 3a, as well as the distinctions among solid, dashed, and dashed-double-dotted edges, one is left with a cyclic graph: there is a cycle that starts at the rightmost middle vertex of the level-1 grouping, follows the blue dashed edge ($--$) to the entry vertex of the level-0 fork-grouping, takes the right branch, and returns along the black solid edge ($-$) to the rightmost middle vertex of the level-1 grouping. However, that path is excluded from consideration because it is not a matched path: the solid edge does not match with the preceding dashed edge.

3.2 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)–(3). To satisfy these more stringent requirements, we define a data structure that is quite different from BDDs (see §3.3 and §4.1).

Requirements (1) and (3) are established in §3.3.4 and §3.5.2, respectively. Requirements (2) and (4) are established in §4 and Appendix §C. Requirement (5) is addressed in §5, §7, and §L; in particular, Tab. 1 at the beginning of §7 lists the fourteen main operations on CFLOBDDs and the asymptotic

⁵The term “call” is by analogy with how matched paths model the actions of procedure calls in graphs used for interprocedural dataflow analysis [65, 68], interprocedural slicing [37], and model checking hierarchical state machines [13, §5].

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. §L 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: Defn. 3.1 (below) and Defn. 4.1 (§4.1). Defn. 3.1 defines the basic structure of CFLOBDDs, whose various elements are depicted in Fig. 3b. Defn. 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 Defn. 3.1, so we postpone introducing the structural invariants until we address canonicity in §4. Where necessary, we distinguish between *mock-CFLOBDDs* (Defn. 3.1) and *CFLOBDDs* (Defn. 4.1), although we typically drop the qualifier “mock-” when there is little danger of confusion. Fig. 3b illustrates Defn. 3.1 using the CFLOBDD that represents Hadamard matrix H_2 .

Definition 3.1 (Mock-CFLOBDD; see Fig. 3b). 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 $0, 1, \dots, k - 1$. The grouping at level k is the *head* of the mock-CFLOBDD.

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

If $i = 0$, g_i is either a *fork grouping* or a *don’t-care grouping*, as depicted in the upper right and lower right of Fig. 3b, respectively. The entry vertex of a level-0 grouping corresponds to a decision point: left branches are for F (or 0); right branches are for T (or 1). A don’t-care grouping has a single exit vertex, and the edges for the left and right branches both connect the entry vertex to the exit vertex. A fork grouping has two exit vertices: the entry vertex’s left and right branches connect the entry vertex to the first and second exit vertices, respectively.

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

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

Fig. 3b shows where the concepts from Defn. 3.1 occur in the CFLOBDD that represents Hadamard matrix H_2 .

3.3.1 An Object-Oriented Pseudo-Code. In later parts of the paper, we state algorithms using an object-oriented pseudo-code. In accordance with the terminology introduced above, the basic classes that are used for representing multi-terminal CFLOBDDs are defined in Fig. 4a: Grouping, InternalGrouping, DontCareGrouping, ForkGrouping, and CFLOBDD. More details about the notation used in our pseudo-code can be found in Appendix §A.

Fig. 4c shows how the CFLOBDD from Fig. 3a 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

```

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)

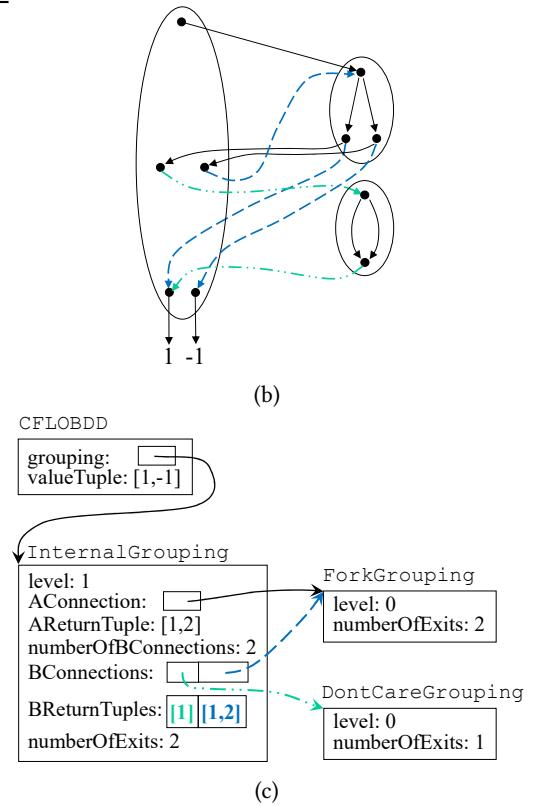


Fig. 4. (a) Datatypes for Grouping, InternalGrouping, DontCareGrouping, ForkGrouping, and CFLOBDD. (b) The CFLOBDD for H_2 (repeated from Fig. 3a). (c) An instance of class CFLOBDD that represents H_2 .

middle vertices, and numbers in the range $[1..numberOfExits]$ serve as exit vertices. In the level-1 **InternalGrouping** in Fig. 4c, 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. Defn. 3.1, Fig. 3b, and Fig. 4a introduce a substantial amount of new terminology. However, the rationale behind it is really quite simple, and goes back to the Matched-Path Principle. In particular, each **InternalGrouping** object g at level $i > 0$ represents a family of matched paths. A traversal of a matched path from g ’s entry vertex to an exit vertex of g uses the fields of g (Fig. 4a)

in the following order:

$$\begin{aligned} \text{matched(at level } i) &= \text{AConnection } \text{matched(at level } i-1) \text{ AReturnTuple}[\cdot] \\ &\quad \text{BConnection } \text{matched(at level } i-1) \text{ BReturnTuples}[\cdot] \end{aligned} \quad (2)$$

Note how Eqn. (2) mimics the form of the grammar for matched paths from Eqn. (1).

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 (Mockproto-CFLOBDD). A *mockproto-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 mockproto-CFLOBDD has the following recursive structure:

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

The difference between a proto-CFLOBDD and a CFLOBDD is that the exit vertices of a proto-CFLOBDD have not been associated with specific values. One cannot argue inductively in terms of CFLOBDDs because its constituents are proto-CFLOBDDs, not full-fledged CFLOBDDs. Thus, to prove that some property holds for a CFLOBDD, there will typically be an inductive argument to establish a property of the proto-CFLOBDD headed by the outermost grouping of the CFLOBDD, with an additional argument about the CFLOBDD’s value edges and terminal values.

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

$$D(0) = 1 \quad D(i) = D(i - 1) + D(i - 1), \quad (3)$$

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

3.3.4 Soundness and an Operational Semantics. Eqn. (3) allows us to establish Requirement (1) from §3.2. Eqn. (3) has the solution $D(i) = 2^i$, so each matched path from the entry vertex of a level- k CFLOBDD passes through the entry vertex of a level-0 grouping exactly 2^k times before reaching a terminal value $v \in V$, for some value domain V . Consequently, each (multi-terminal) CFLOBDD represents a function in $\{T, F\}^{2^k} \rightarrow V$ —i.e., the same set of functions that decision trees represent.

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

Subroutine `InterpretGrouping` performs a recursive traversal over n , following AConnections, BConnections, and return edges. When a level-0 grouping is reached, the value of the current Boolean variable is consulted (line [8], in the case of a ForkGrouping), or ignored (line [9], in the case of a DontCareGrouping). (Line [10] can be ignored for now; it is an optimization that is discussed in §3.5.2.) 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

Algorithm 1: An operational semantics of CFLOBDDs

```

1 Algorithm InterpretCFLOBDD( $n, a$ )
2   Input: CFLOBDD  $n$ , Assignment  $a[1..2^{n.\text{grouping.level}}]$ 
3   Output: A value in the range of the function represented by  $n$ 
4   begin
5     | return valueTuple[InterpretGrouping( $n.\text{grouping}, a$ )];
6   end
7
8 SubRoutine InterpretGrouping( $g, a$ )
9   Input: Grouping  $g$ , Assignment  $a[1..2^{g.\text{level}}]$ 
10  Output: An unsigned integer
11  begin
12    | if  $g == \text{ForkGrouping}$  then return  $1 + a[1]$                                 //  $F \mapsto 1; T \mapsto 2;$ 
13    | if  $g == \text{DontCareGrouping}$  then return  $1$                                          //  $F, T \mapsto 1;$ 
14    | if  $g == \text{NoDistinctionProtoCFLOBDD}(g.\text{level})$  then return  $1$           //  $F, T \mapsto 1;$ 
15    | Assignment  $a_A = a[1, 2^{g.\text{level}-1}]$ ;
16    | Assignment  $a_B = a[2^{g.\text{level}-1} + 1, 2^{g.\text{level}}]$ ;
17    | unsigned int  $i = \text{InterpretGrouping}(g.\text{AConnection}, a_A)$ ;
18    | unsigned int  $k = \text{InterpretGrouping}(g.\text{BConnections}[i], a_B)$ ;
19    | return  $g.\text{BReturnTuples}[1](k)$ ;
20  end
21
22 end

```

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 . Fig. 3a is a multi-terminal CFLOBDD; the level-1 grouping has two exit vertices that are mapped to 1 and -1 .

Groupings that have more than two exit vertices naturally arise in the interior groupings of CFLOBDDs—even in Boolean-valued CFLOBDDs. For instance, a level- $i-1$ grouping used as an A-connection can 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. Fig. 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

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

Consider how Fig. 6a 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

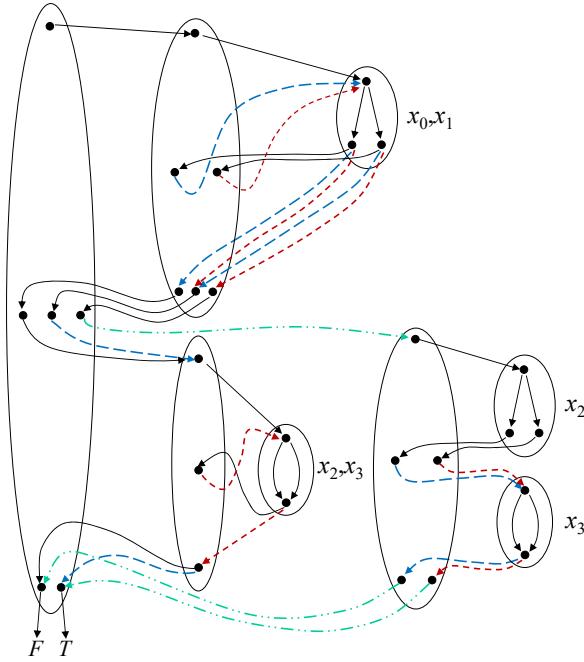


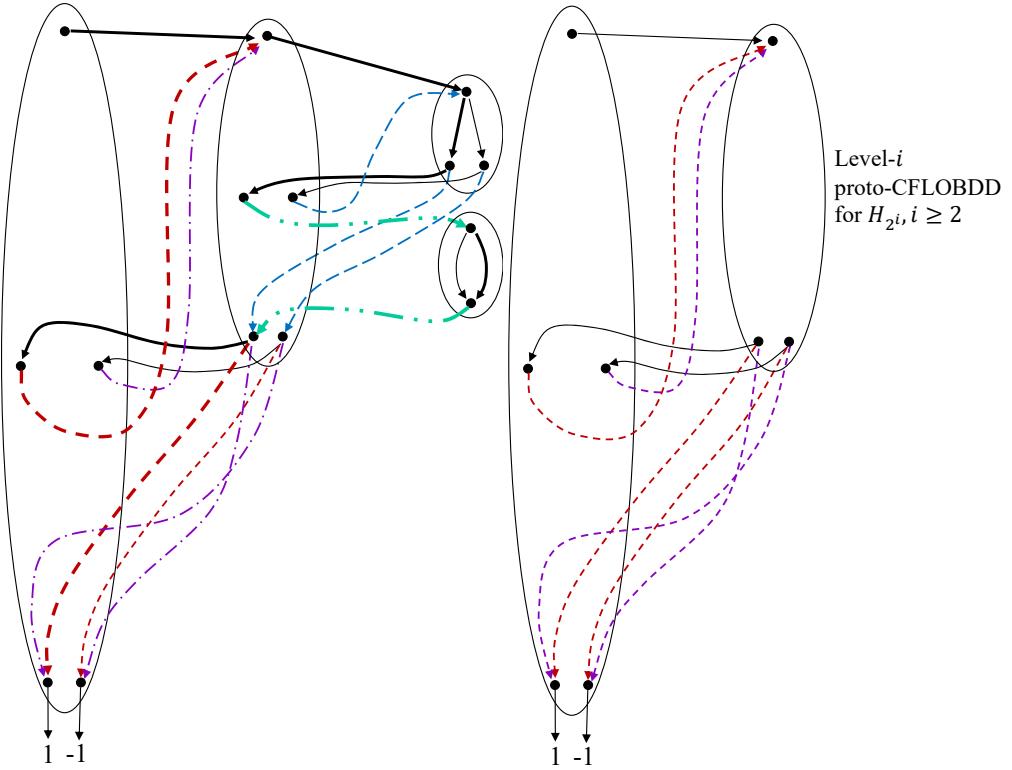
Fig. 5. CFLOBDD for the Boolean function $\lambda x_0 x_1 x_2 x_3. (x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. (For clarity, some of the level-0 groupings have been duplicated.)

starts from the level-2 grouping's entry vertex. It goes to the level-1 grouping's entry vertex, where $[x_0 \mapsto F, y_0 \mapsto T]$ is interpreted as for H_2 —i.e., the first occurrence of H_2 in “ $H_2 \otimes H_2$ ”—in this case, returning to the leftmost middle vertex of the level-2 grouping. At this point, the path follows the red dashed edge back to the level-1 grouping's entry vertex, where $[x_1 \mapsto F, y_1 \mapsto T]$ is interpreted as for H_2 —the second occurrence of H_2 in “ $H_2 \otimes H_2$.” The path then follows the matching red dashed return edge to the leftmost exit vertex of the level-2 grouping, and reaches terminal value 1.

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

The continuation of this pattern gives an inductive construction of the CFLOBDDs for the other members of \mathcal{H} . Given the level- i CFLOBDD for H_{2^i} , $i \geq 2$, $H_{2^{i+1}} = H_{2^i} \otimes H_{2^i}$ is created by introducing a new outermost grouping at level $i + 1$, again with five vertices and nine edges. (See Fig. 6b.) The same pattern of “calls” is used for the A- and B-connections and their return edges: each matched path makes two sequential invocations of the level- i grouping for H_{2^i} . In other words,

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



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

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 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:

$$P(0) = 2 \quad P(i) = \sum_j A_j(i-1) \cdot P(i-1). \quad (4)$$

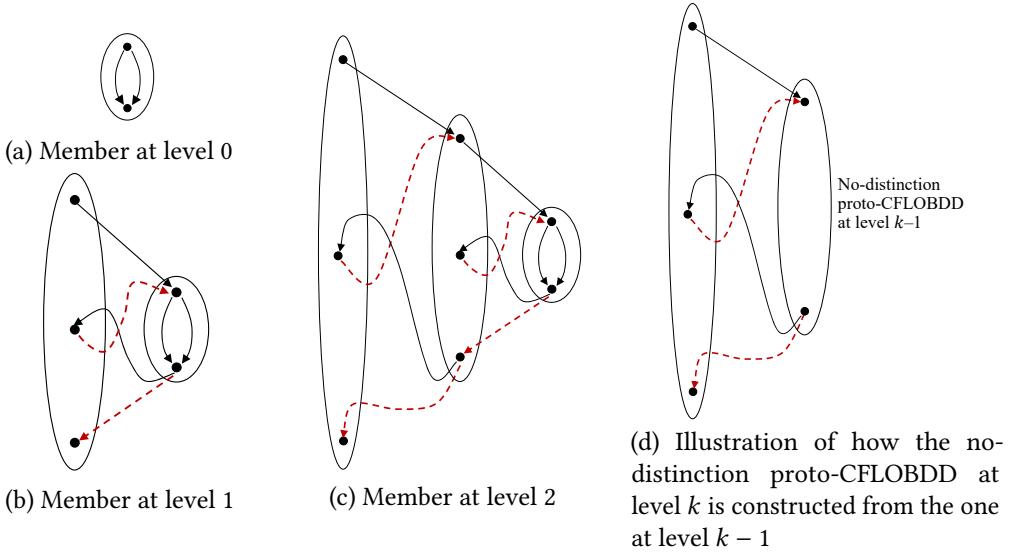


Fig. 7. The family of no-distinction proto-CFLOBDDs.

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 Eqn. (4) can be rewritten as

$$P(i) = P(i - 1) \cdot P(i - 1), \quad (5)$$

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 (Eqn. (5)). Consequently, a CFLOBDD at level i has 2^{2^i} matched paths.

3.5.2 Best-Case Compression: No-Distinction Proto-CFLOBDDs. Fig. 7a, 7b, and 7c show the first three members of a family of proto-CFLOBDDs that often arise as sub-structures of CFLOBDDs: the single-entry/single-exit proto-CFLOBDDs of levels 0, 1, and 2, respectively. Because every matched path through each of these structures ends up at the unique exit vertex of the highest-level grouping, there is no “decision” to be made during each visit to a level-0 grouping. In essence, as we work our way through such a structure during the interpretation of an assignment, the value assigned to each argument variable makes no difference.

We call this family the *no-distinction proto-CFLOBDDs*. Fig. 7d 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 $\lambda x_0, x_1, \dots, x_{2^k-1}.F$ 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 $\lambda x_0, x_1, \dots, x_{2^k-1}.T$, 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	$k + 1$	$3k + 2$	$4k + 3$

^aThe height of a CFLOBDD is the level of the outermost grouping.

The best-case CFLOBDD size—whether measured in the number of groupings, vertices, or edges—grows linearly with the level of the outermost grouping, which is *logarithmic* in the number of Boolean variables. In contrast, decision trees grow *exponentially* in the number of Boolean variables. These observations show that Requirement (3) from §3.2 is met: in the best case, a decision tree of height 2^k and size 2^{2^k} can be encoded as a level- k CFLOBDD of size k .

Remark. Because the family of no-distinction proto-CFLOBDDs is so compact, in designing CFLOBDDs we did not feel the need to mimic the “ply-skipping transformation” of Reduced OBDDs (ROBDDs) [16, 17], in which “don’t-care” nodes are removed from the representation. In ROBDDs, in addition to reducing the size of the data structure, the chief benefit of ply-skipping is that operations can skip over levels in portions of the data structure in which no distinctions among variables are made. Essentially the same benefit is obtained by having the algorithms that process CFLOBDDs carry out appropriate special-case processing when no-distinction proto-CFLOBDDs are encountered. Such processing is carried out, for instance, in line [10] of Alg. 1: in InterpretCFLOBDD(), when Grouping g is the head of a NoDistinctionProtoCFLOBDD, both g and the entire Assignment a can be ignored because g has only a single exit vertex.

Whereas in the best case, the CFLOBDD for a function f can be double-exponentially smaller than the decision tree for f , ROBDDs are incapable of such a degree of compression. *Quasi-reduced BDDs* are the version of BDDs in which don’t-care nodes are *not* removed (i.e., plies are not skipped), and thus all paths from the root to a terminal value have length n , where n is the number of variables. The size of a quasi-reduced BDD is at most a factor of $n + 1$ larger than the size of the corresponding ROBDD [75, Thm. 3.2.3]. Thus, although ROBDDs can give better-than-exponential compression compared to decision trees, what one has is not double-exponential compression: at best, it is linear compression of exponential compression. Moreover, in §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

⁶The family of no-distinction proto-CFLOBDDs from Fig. 7 is one such family G .

- (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-1} , and (ii) level- $(i-1)$ groupings used in the CFLOBDD for g_{i-1} .⁷

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

Moreover, in all cases encountered to date, it is possible to give an explicit algorithm for constructing the 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 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 CANONICALNESS

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

4.1 CFLOBDDs Defined, Part II: Additional Structural Invariants

As described in §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 Fig. 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 Fig. 5,

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

⁷Condition 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, \dots, G_l that each grow linearly in the level i .

Rationale. The structural invariants are designed to ensure that—for a given order on the Boolean variables—each Boolean function has a unique, canonical representation as a CFLOBDD. In reading Defn. 4.1 below, it will help to keep in mind that the goal of the invariants is to force there to be a *unique* way to fold a given decision tree into a CFLOBDD that represents the same Boolean function. The decision-tree folding method is discussed in §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 mockproto-CFLOBDD (Defns. 3.1 and 3.2) in which every grouping/proto-CFLOBDD in n satisfies the *structural invariants* given below. In particular, let c be an A -connection edge or B -connection edge from grouping g_i to g_{i-1} , with associated return tuple rt_c .

- (1) If c is an A -connection, then rt_c must map the exit vertices of g_{i-1} one-to-one, and in order, onto the middle vertices of g_i : Given that g_{i-1} has k exit vertices, there must also be k middle vertices in g_i , and rt_c must be the k -tuple $[1, 2, \dots, k]$. (That is, when rt_c is considered as a map on indices of exit vertices of g_{i-1} , rt_c is the identity map.)
- (2) If c is the B -connection edge whose source is middle vertex $j + 1$ of g_i and whose target is g_{i-1} , then rt_c must meet two conditions:
 - (a) It must map the exit vertices of g_{i-1} one-to-one (but not necessarily onto) the exit vertices of g_i . (That is, there are no repetitions in rt_c .)
 - (b) It must “compactly extend” the set of exit vertices in g_i defined by the return tuples for the previous j B -connections: Let $rt_{c_1}, rt_{c_2}, \dots, rt_{c_j}$ 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 $rt_{c_1}, rt_{c_2}, \dots, rt_{c_j}$, and let n be the largest value in S . (That is, n is the index of the rightmost exit vertex of g_i that is a target of any of the return tuples $rt_{c_1}, rt_{c_2}, \dots, rt_{c_j}$.) If S is empty, then let n be 0.

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

- (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.

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. §5.1 discusses how hash-consing [31] 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 §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.

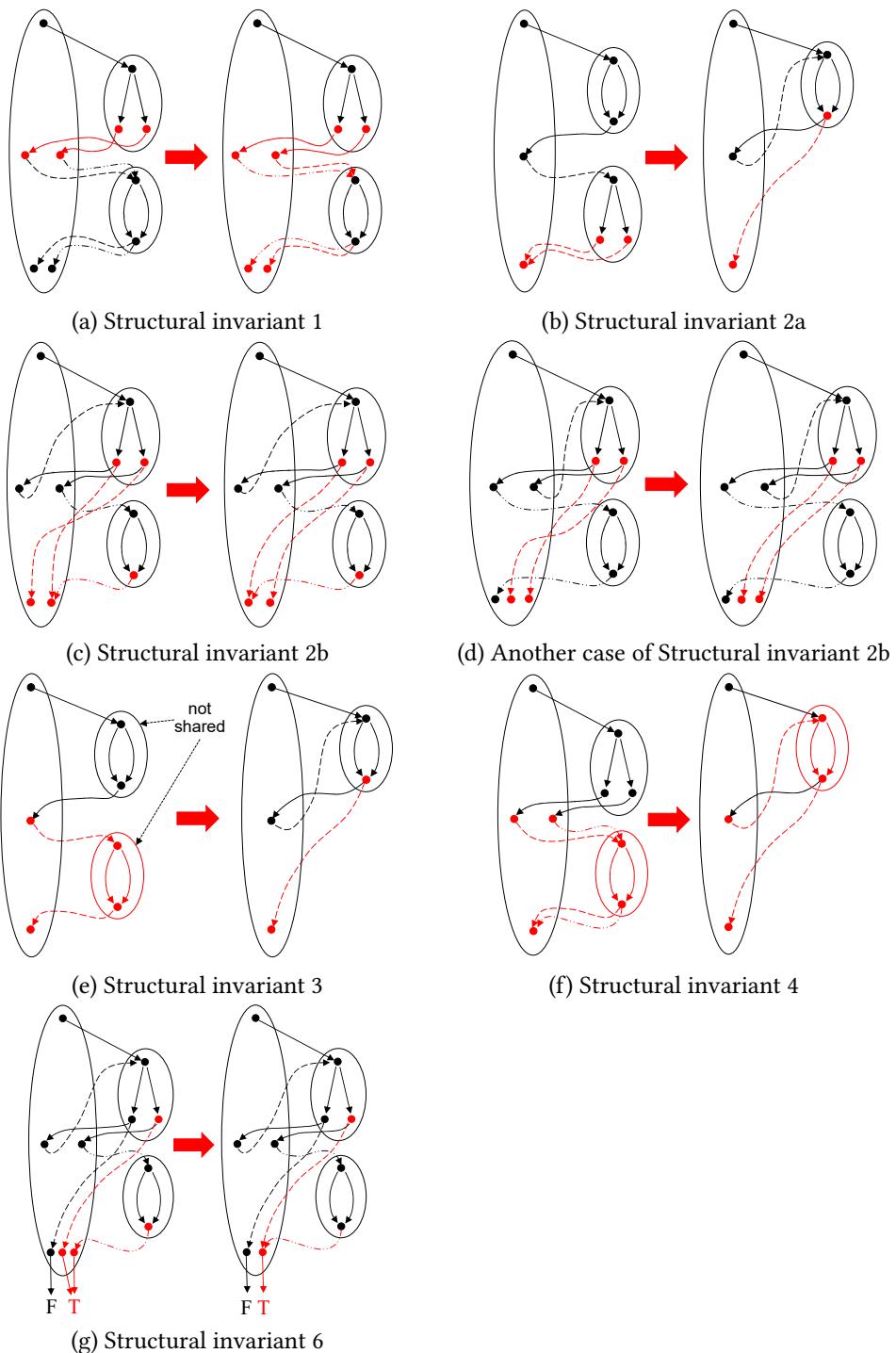


Fig. 8. To the left of each arrow, a mock-proto-CFLOBDD that violates the indicated structural invariant; to the right, a corrected proto-CFLOBDD. Invariant violations and their rectifications are shown in red.

Fig. 8 illustrates structural invariants 1, 2a, 2b, 3, 4, and 6. In each case, a mockproto-CFLOBDD that violates one of the structural invariants is shown on the left, and an equivalent proto-CFLOBDD that satisfies the structural invariants is shown on the right.

The CFLOBDD from Fig. 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 $[1, 2]$.

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

Also in Fig. 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; i.e., the corresponding return tuple is $[1, 2, 3]$.

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

One artifact of the greedy, left-to-right decision-tree folding method used in §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 $ex_C = s$. \square*

The proof of Prop. 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, Prop. 4.1 elucidates a semantic consequence of the structural invariants.⁹

Example 4.2. Prop. 4.1 can be illustrated using Fig. 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 Fig. 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 $[1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2]$, so s is $[1, 2]$.

4.2 Canonicity of CFLOBDDs

CFLOBDDs are a canonical representation of functions over Boolean arguments, i.e., each decision tree with 2^{2^k} leaves is represented by exactly one isomorphism class of level- k CFLOBDDs. (The notion of isomorphism of CFLOBDDs was introduced in footnote 8.)

⁹§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 §C, Prop. 4.1 is used in the proof of Prop. C.1, which establishes property (3) from §4.2.

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 Obs. 2.1, and argue not in terms of Boolean functions but in terms of *representations* of Boolean functions—specifically, we relate two kinds of Boolean-function representations

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

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

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

The proof that CFLOBDDs are a canonical representation of Boolean functions is in Appendix §C. We already showed that Obligation 1 is satisfied in §3.3.4.

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

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

Note that Obligation 1 and 2 are exactly Requirements (1) and (2) from §3.2, respectively. Moreover, Obligations 1–3 together show that Requirement (4) from §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 [31] enforces the invariant that only a single representative exists for each value constructed from some datatype. Hash-consing should not be confused with canonicity (§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-consing¹⁰ satisfies an even stronger form of equivalence. In particular, Thm. 4.3 can be restated to read “... then C_1 and C_2 are identical.”

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 §7 that create InternalGroupings, such as PairProduct (Alg. 11) and Reduce (Alg. 13), have the following form:

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

The operation NoDistinctionProtoCFLOBDD (Alg. 3), which constructs the members of the family of no-distinction proto-CFLOBDDs depicted in Fig. 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 (Alg. 2) illustrates the use of RepresentativeCFLOBDD: ConstantCFLOBDD(k, v) returns a hash-consed CFLOBDD that represents a constant function of the form $\lambda x_0, x_1, \dots, x_{2^k-1}.v$.

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 mockproto-CFLOBDDs—i.e., structural invariants (1)–(4) of Defn. 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 Alg. 13 of §7.3), 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 mockproto-CFLOBDDs that violate the proto-CFLOBDD structural invariants.

5.2 Equality Testing for CFLOBDDs and proto-CFLOBDDs

As discussed in §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

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

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

5.3 Function Caching

A function cache (or *memo function* [54]) for a function F is an associative-lookup table—typically a hash table—of pairs of the form $[x, F(x)]$, keyed on the value of x . The table is consulted each time F is applied to some argument, and updated after a return value is computed for a never-before-seen argument. The technique saves the cost of re-performing the computation of F for an argument on which F has previously been called, at the expense of performing a lookup on F 's argument at the beginning of each call. Our implementation of CFLOBDDs uses function caching for a number of the operations described in the remainder of the paper, such as PairProduct (Alg. 11) and Reduce (Alg. 13). 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  $\text{cache}_F(x) \neq \text{NULL}$  return  $\text{cache}_F(x)$ ;
    ...
     $\text{cache}_F(x) = \text{retVal}$ ; // Update the cache with the return value
    return  $\text{retVal}$ ;
}
```

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 (§5.2) also improves the performance of function caching.

6 A DENOTATIONAL SEMANTICS

In §3.3.4, we gave an operational semantic definition of the function that a CFLOBDD represents. In this section, we give denotational semantics, which defines the function that a CFLOBDD denotes. In particular, this semantics associates the i^{th} element of a CFLOBDD's valueTuple with the set of Assignments (i.e., language of bit-strings) that map to the i^{th} element. That is, a CFLOBDD n at level k denotes a function

$$\llbracket n \rrbracket : [1..|n.\text{valueTuple}|] \rightarrow \mathcal{P}(\{0, 1\}^{2^k}).$$

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

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

$$\llbracket g_{Fork} \rrbracket \stackrel{\text{def}}{=} [\{0\}, \{1\}] \quad \llbracket g_{DC} \rrbracket \stackrel{\text{def}}{=} [\{0, 1\}].$$

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

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

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

$$\llbracket g \rrbracket_{1 \times m} = \begin{cases} [\{0\}, \{1\}]_{1 \times 2} & \text{if } g = \text{ForkGrouping} \\ [\{0, 1\}]_{1 \times 1} & \text{if } g = \text{DontCareGrouping} \\ \llbracket g.AConnection \rrbracket_{1 \times p} \times \\ \left[\begin{array}{c} \vdots \\ \llbracket g.BConnections[i] \rrbracket_{1 \times k_i} \times \llbracket g.BReturnTuples[i] \rrbracket_{k_i \times m} \\ \vdots \\ \end{array} \right]_{p \times m} & \text{otherwise} \\ i \in \{1..p\} \end{cases}$$

Finally, for a CFLOBDD n , $\llbracket n \rrbracket$ is defined as follows:

$$\llbracket n \rrbracket_{1 \times |n.valueTuple|} \stackrel{\text{def}}{=} \llbracket n.grouping \rrbracket_{1 \times |n.grouping.numberOfExits|}$$

Example 6.1. For the five proto-CFLOBDDs depicted in Fig. 9, the vectors of languages are as follows (read top-to-bottom by level):

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

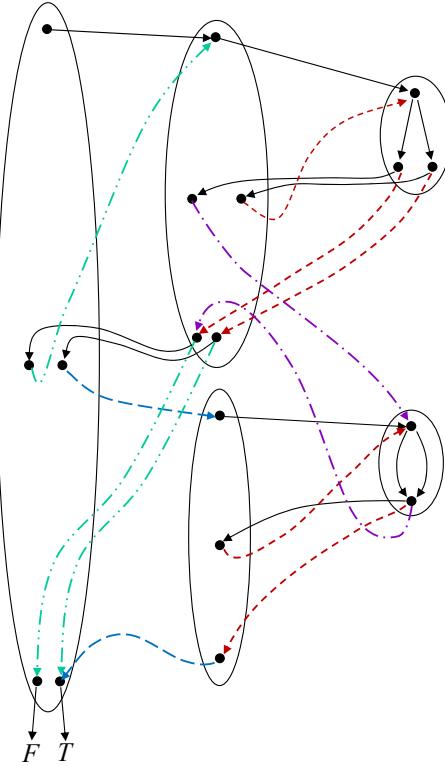


Fig. 9. CFLOBDD representation of the function $\lambda w, x, y, z. (w \wedge x) \vee (y \wedge z)$, with variable ordering $\langle w, x, y, z \rangle$

Algorithm 2: ConstantCFLOBDD

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

```

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

7 ALGORITHMS ON CFLOBDDS

In this section, we describe operations to construct or combine CFLOBDDs. To aid the reader, Tab. 1 lists the fourteen main operations on CFLOBDDs, together with references to where the algorithm for each operation is presented (and where it is discussed), along with each operation's asymptotic running time and the asymptotic running time of the analogous BDD operation. Readers familiar with BDDs will find that the algorithms for operations on CFLOBDDs are somewhat more complicated than their BDD counterparts, mainly due to the need to maintain the CFLOBDD structural invariants (Defn. 4.1).

7.1 Primitive CFLOBDD-Creation Operations

Operation	Type Signature	Description	Time Complexity	
			CFLOBDD	BDD
Equal (§5.2)	CFLOBDD × CFLOBDD → Boolean	Checks if two CFLOBDDs are equal	$O(1)$	$O(1)$
ConstantCFLOBDD (Alg. 2, §7.1.1)	Int(k) × Value(v) → CFLOBDD	Creates a CFLOBDD for a constant function $\lambda x_0 \dots x_{k-1}.v$	$O(k)$	$O(2^k)$
FalseCFLOBDD (Alg. 4, §7.1.1)	Int(k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \dots x_{k-1}.F$	$O(k)$	$O(2^k)$
TrueCFLOBDD (Alg. 5, §7.1.1)	Int(k) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \dots x_{k-1}.T$	$O(k)$	$O(2^k)$
NoDistinctinctionProtoCFLOBDD (Alg. 3, §7.1.1)	Int(k) → Proto-CFLOBDD	Creates a NoDistinctinctionProtoCFLOBDD for 2^k variables	$O(k)$	N/A
ProjectionCFLOBDD (Alg. 6, §7.1.2)	Int(k) × Int(i) → CFLOBDD	Creates a CFLOBDD for the function $\lambda x_0 \dots x_{k-1}.x_i$	$O(k)$	$O(2^k)$
FlipValueTupleCFLOBDD (Alg. 7, §7.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are flipped	$O(1)$	$O(c _B)$
ComplementCFLOBDD (Alg. 7, §7.2.1)	CFLOBDD(c) → CFLOBDD	Creates a CFLOBDD such that the output values are complemented	$O(1)$	$O(c _B)$
ScalarMultiplyCFLOBDD (Alg. 8, §7.2.2)	CFLOBDD(c) × Value(v) → CFLOBDD	Performs $c' = c * v$	$O(c \times c')$	$O(c _B)$
BinaryApplyAndReduce (Alg. 10, §7.3)	CFLOBDD(c_1) × CFLOBDD(c_2) × Operation $op \rightarrow$ 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 (Alg. 21, §7.8.1)	CFLOBDD(c) → CFLOBDD	Computes the number of paths to every exit vertex of every grouping	$O(c)$	$O(c _B)$ (See §10.1.2.)
Sampling (Alg. 22, §7.8.2)	CFLOBDD(c) → String	Samples a path from c	$O(\max(\text{vars}, c))$	$O(\max(\text{vars}, c _B))$ (See §10.1.2.)
KroneckerProduct (App. §G & Alg. 17, §7.5)	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 (Algs. 19 and 20, §7.7)	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	$O(N^3)$

Table 1. List of operations on CFLOBDDs; vars denotes the number of Boolean variables ($= 2^k$, where k is the number of levels of the CFLOBDD). The size measure $| \cdot |$ counts the number of groupings, vertices, and edges—with no double-counting of shared groupings due to hash-consing. In the column for the time complexities of BDD operations, an occurrence of c refers to a BDD argument of the operation, and $|c|_B$ denotes the size of BDD c (the number of nodes and edges). For quasi-reduced BDDs, the time to construct the analog of NoDistinctinctionProtoCFLOBDD 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 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.ARearTuple = [1];
8   g.numberOfBConnections = 1;
9   g.BConnections[1] = g.AConnection;
10  g.BReturnTuples[1] = [1];
11  g.numberOfExits = 1;
12  return RepresentativeGrouping(g);
13 end
```

Algorithm 4: FalseCFLOBDD

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

```

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

Algorithm 5: TrueCFLOBDD

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

```

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

7.1.1 Constant Functions. The CFLOBDD-creation operation ConstantCFLOBDD, given as Alg. 2, produces the family of CFLOBDDs that represent functions of the form $\lambda x_0, x_1, \dots, x_{2^k-1}.v$, where v is some constant value. ConstantCFLOBDD(k, v) uses as a subroutine NoDistinctionProtoCFLOBDD (Alg. 3), which constructs the no-distinction proto-CFLOBDD for a given level k (see also Fig. 7). ConstantCFLOBDD can be used to construct CFLOBDDs for the constant functions $\lambda x_0, x_1, \dots, x_{2^k-1}.F$ (Alg. 4) and $\lambda x_0, x_1, \dots, x_{2^k-1}.T$ (Alg. 5). ConstantCFLOBDD(k, v) runs in time $O(k)$ and uses at most $O(k)$ space.

7.1.2 Projection Functions. A second family of CFLOBDD-creation operations produces the Boolean-valued (*single-variable*) *projection functions* of the form $\lambda x_0, x_1, \dots, x_{2^k-1}.x_i$, where i ranges from 0 to $2^k - 1$. Fig. 10 illustrates the structure of the CFLOBDDs that represent these functions. Alg. 6 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.

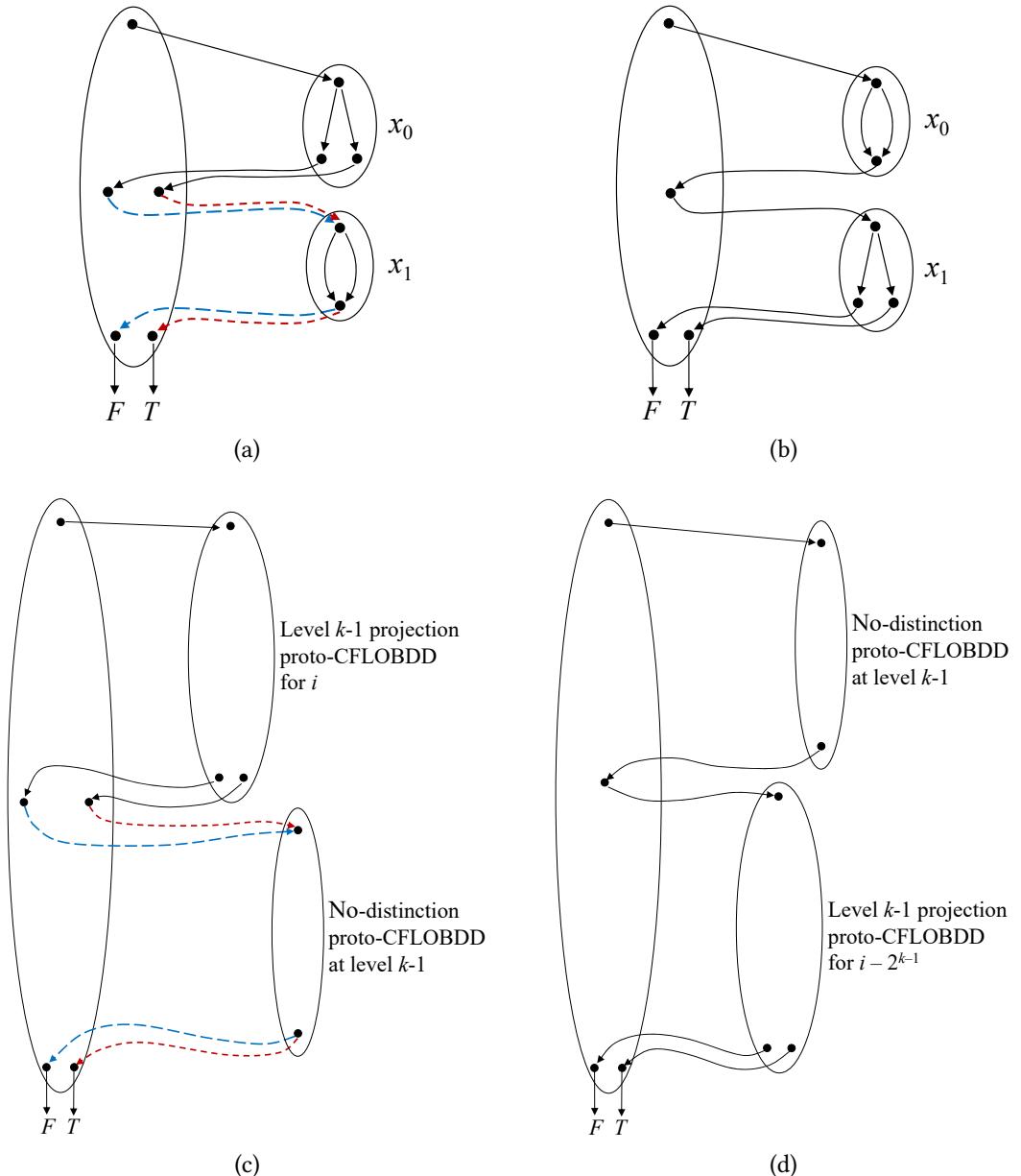


Fig. 10. (a) CFLOBDD for $\lambda x_0 x_1.x_0$; (b) CFLOBDD for $\lambda x_0 x_1.x_1$; (c) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \dots, x_{2k-1}.x_i$, when $0 \leq i < 2^{k-1}$; (d) schematic drawing of CFLOBDDs that represent projection functions of the form $\lambda x_0, x_1, \dots, x_{2k-1}.x_i$, when $2^{k-1} \leq i < 2^k$.

7.2 Unary Operations on CFLOBDDs

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

Algorithm 6: ProjectionProtoCFLOBDD

```

1 Algorithm ProjectionCFLOBDD( $k, i$ )
   Input: int  $k$  (level), int  $i$  (index)
   Output: CFLOBDD representing function  $\lambda x_0, x_1, \dots, x_{2^k-1}.x_i$ 
2 begin
3   assert( $0 \leq i < 2^{**k}$ );
4   return RepresentativeCFLOBDD(ProjectionProtoCFLOBDD( $k, i$ ), [F,T]);
5 end
6 end
1 SubRoutine ProjectionProtoCFLOBDD( $k, i$ )
   Input: int  $k$  (level), int  $i$  (index)
   Output: Grouping  $g$  representing function  $\lambda x_0, x_1, \dots, x_{2^k-1}.x_i$ 
2 begin
3   if  $k == 0$  then  $// i$  must also be 0
4     return RepresentativeForkGrouping;
5   else
6     InternalGrouping  $g =$  new InternalGrouping( $k$ );
7     if  $i < 2^{**k-1}$  then  $// i$  falls in AConnection range
8        $g.AConnection =$  ProjectionProtoCFLOBDD( $k-1, i$ );
9        $g.ARewriteTuple = [1, 2];$ 
10       $g.numBConnections = 2;$ 
11       $g.BConnection[1] =$  NoDistinctionProtoCFLOBDD( $k-1$ );
12       $g.BReturnTuples[2] = [1];$ 
13       $g.BConnections[2] = g.BConnection[1];$ 
14       $g.BReturnTuples[2] = [2];$ 
15       $g.numberOfExits = 2;$ 
16    else  $// i$  falls in BConnection range
17       $g.AConnection =$  NoDistinctionProtoCFLOBDD( $k-1$ );
18       $g.ARewriteTuple = [1];$ 
19       $g.numBConnections = 1;$ 
20       $i = i - 2^{**k-1};$   $// Remove high-order bit for recursive call$ 
21       $g.BConnections[1] =$  ProjectionProtoCFLOBDD( $k-1, i$ );
22       $g.BReturnTuples[1] = [1, 2];$ 
23       $g.numberOfExits = 2;$ 
24    end
25    return RepresentativeGrouping( $g$ );
26  end
27 end
28 end

```

7.2.1 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

Algorithm 7: 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 8: ScalarMultiplyCFLOBDD

```

Input: CFLOBDD c, Value v
Output: CFLOBDD c' = c * v
1 begin
2   | // Multiply CFLOBDD c by the CFLOBDD for the constant function  $\lambda x_0, x_1, \dots, x_{2^k-1}.v$ 
3   | return BinaryApplyAndReduce(c, ConstantCFLOBDD(c.level, v), (op)Times); // (See §7.3)
3 end

```

be used to implement the operation `ComplementCFLOBDD`, which forms the Boolean complement of its argument, in an efficient manner. The pseudocode for these functions is given in Alg. 7. `FlipValueTupleCFLOBDD` and `ComplementCFLOBDD` are constant-time operations.

7.2.2 Scalar Multiplication. Function `ScalarMultiplyCFLOBDD` of Alg. 8 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 $\lambda x_0, x_1, \dots, x_{2^k-1}.v$, which is multiplied by CFLOBDD *c* using `BinaryApplyAndReduce`—the generic operation for binary CFLOBDD operations (discussed in §7.3)—with the multiplication operator `Times` passed as the third argument.

7.3 Binary Operations on CFLOBDDs

This section presents an algorithm for performing binary operations on CFLOBDDs. The algorithm is parameterized in terms of a binary operation `op` that is to be applied pointwise to the range values of two CFLOBDDs. That is, given the CFLOBDDs for two functions *n*₁ and *n*₂ and binary operation `op`, the goal of the algorithm is to create the CFLOBDD for *n*₁ op *n*₂ where, for each assignment

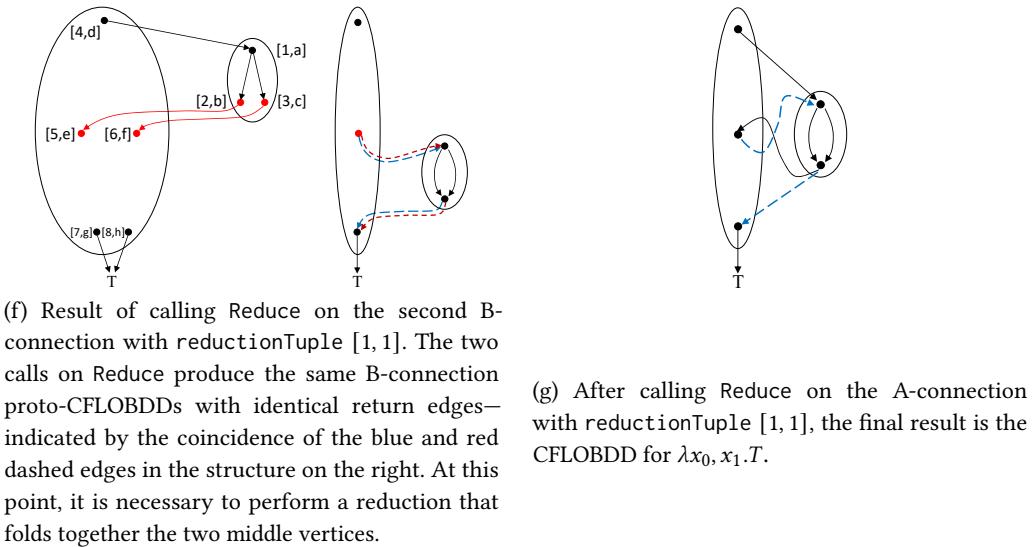
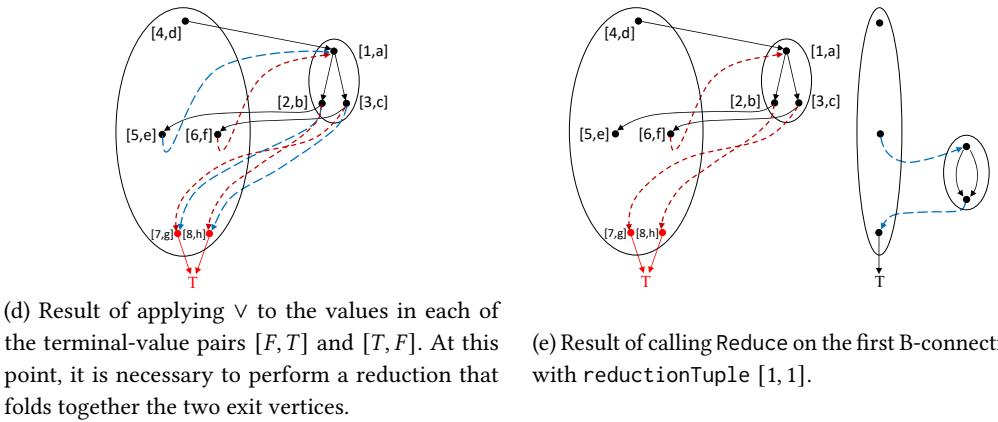
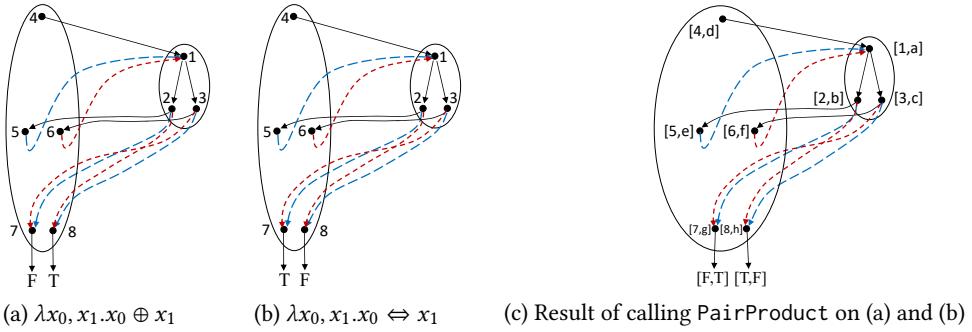


Fig. 11. Illustration of how $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$ results in $\lambda x_0, x_1.T$.

Algorithm 9: CollapseClassesLeftmost

Input: Tuple equivClasses
Output: Tuple \times Tuple [projectedClasses, renumberedClasses]

```

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

$a, (n_1 \text{ op } n_2)(a) = n_1(a) \text{ op } n_2(a)$. Operation op could be $+, -, *, /$, etc., or—if the functions are Boolean-valued— \vee, \wedge, \oplus , etc. As with BDDs, such operations on CFLOBDDs can be implemented via a two-step process¹¹

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

Just as there can be multiple occurrences of a given node in a BDD, there can be multiple occurrences of a given grouping in a CFLOBDD. To avoid a blow-up in costs, binary operations need to avoid making repeated calls on a given pair of groupings $g_1 \in n_1$ and $g_2 \in n_2$. Assuming that the hash-table lookup and insertion methods used for hash-consing (§5.1) and function caching (§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—i.e., $O(|n_1| \times |n_2|)$.¹² §L 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 $n_1 \text{ op } n_2$. Consequently, binary operations satisfy Requirement (5); i.e., they run in (expected) time that is polynomial in the sizes of the input and output CFLOBDDs.

Fig. 11 illustrates this method by showing how the CFLOBDD for $\lambda x_0, x_1.T$ is obtained as the result of a Boolean- \vee : $(\lambda x_0, x_1.x_0 \oplus x_1) \vee (\lambda x_0, x_1.x_0 \Leftrightarrow x_1)$. Fig. 11c shows the result of the product construction (PairProduct, Alg. 11). Fig. 11d, e, f, and g illustrate some of the steps of the reduction algorithm (Reduce, Alg. 13). Fig. 11 is discussed in more detail in Ex. 7.1.

The algorithms involved are given as Algs. 9–14. (In Algs. 10 and 11, we assume that the CFLOBDD or Grouping arguments are objects whose highest-level groupings are all at the same level.)

- The operation BinaryApplyAndReduce, given as Alg. 10, starts with a call on PairProduct (line [2]). PairProduct, given as Algs. 11 and 12, performs a recursive traversal of the two Grouping arguments, g_1 and g_2 , 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 ,

¹¹The two-step process is conceptual for BDDs: the two steps can be combined in an implementation (e.g., see [26, §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 Ex. 7.1.

¹²More 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 $\sum_{k=0}^l \sum \{|g_1| \times |g_2| \mid g_1 \in n_1 \uparrow gr[k] \text{ and } g_2 \in n_2 \uparrow gr[k]\}$.

Algorithm 10: BinaryApplyAndReduce

Input: CFLOBDDs n_1, n_2 and Operation op
Output: CFLOBDD $n = n_1 \text{ op } n_2$

```

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

```

Algorithm 11: PairProduct

Input: Groupings g_1, g_2
Output: Grouping g : product of g_1 and g_2 ; PairTuple ptAns: tuple of pairs of exit vertices

```

1 begin
2   if  $g_1$  and  $g_2$  are both no-distinction proto-CFLOBDDs then return [  $g_1, [[1,1]]$  ];
3   if  $g_1$  is a no-distinction proto-CFLOBDD then return [  $g_2, [[1,k] : k \in$ 
4      $[1..g_2.\text{numberOfExits}]]$  ];
5   if  $g_2$  is a no-distinction proto-CFLOBDD then return [  $g_1, [[k,1] : k \in$ 
6      $[1..g_1.\text{numberOfExits}]]$  ];
7   if  $g_1$  and  $g_2$  are both fork groupings then return [  $g_1, [[1,1],[2,2]]$  ];
8   // Pair the A-connections
9   Grouping×PairTuple [gA,ptA] = PairProduct( $g_1.\text{AConnection}$ ,  $g_2.\text{AConnection}$ );
10  InternalGrouping g = new InternalGrouping( $g_1.\text{level}$ );
11  g.ACConnection = gA ;
12  g.ARReturnTuple = [1..|ptA|];                                // Represents the middle vertices
13  g.numberOfBConnections = |ptA| ;
14  // Continued in Alg. 12
15 end

```

a descriptor of the exit vertices of g in terms of pairs of exit vertices of the highest-level groupings of g_1 and g_2 . (See Alg. 11, lines [2]–[5] and Alg. 12, lines [20]–[38].)

From the semantic perspective, each exit vertex e_1 of g_1 represents a (non-empty) set A_1 of variable-to-Boolean-value assignments that lead to e_1 along a matched path in g_1 ; similarly, each exit vertex e_2 of g_2 represents a (non-empty) set of variable-to-Boolean-value assignments A_2 that lead to e_2 along a matched path in g_2 . If pt , the descriptor of g ’s exit

Algorithm 12: PairProduct (cont.)

```

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

```

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 (§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 [34]). Similarly, the number of middle vertices in g is bounded by the product of the numbers of middle vertices in g_1 and g_2 (see line [10]). Thus, the size of g is bounded by the product of the sizes of g_1 and g_2 . Consequently, the cost of the call on `PairProduct` in line [2] of Alg. 10 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]–[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 13: Reduce

Input: Grouping g , ReductionTuple $reductionTuple$

Output: Grouping g' that is "reduced"

```

1 begin
2   // Test whether any reduction actually needs to be carried out
3   if  $reductionTuple == [1../reductionTuple]$  then
4     | return  $g$ ;
5   end
6   // If only one exit vertex, then collapse to no-distinction
7   // proto-CFLOBDD
8   if  $\{|x : x \in reductionTuple|\} == 1$  then
9     | return NoDistinctionProtoCFLOBDD( $g.level$ );
10    end
11   InternalGrouping  $g' = new InternalGrouping(g.level)$ ;
12    $g'.numberOfExits = |\{x : x \in reductionTuple\}|$ ;
13   Tuple  $reductionTupleA = []$ ;
14   for  $i \leftarrow 1$  to  $g.numberOfBConnections$  do
15     | Tuple deducedReturnClasses =  $[reductionTuple(v) : v \in g.BReturnTuples[i]]$ ;
16     | Tuple $\times$ Tuple [ $inducedReturnTuple$ ,  $inducedReductionTuple$ ] =
17       CollapseClassesLeftmost(deducedReturnClasses);
18     | Grouping  $h = Reduce(g.BConnection[i], inducedReturnTuple)$ ;
19     | int position = InsertBConnection( $g'$ ,  $h$ ,  $inducedReturnTuple$ );
20     |  $reductionTupleA = reductionTupleA \parallel position$ ;
21   end
22   Tuple $\times$ Tuple [ $inducedReturnTuple$ ,  $inducedReductionTuple$ ] =
23   CollapseClassesLeftmost( $reductionTupleA$ );
24   Grouping  $h' = Reduce(g.AConnection, inducedReductionTuple)$ ;
25    $g'.AConnection = h'$ ;
26    $g'.AReturnTuple = inducedReturnTuple$ ;
27   return RepresentativeGrouping( $g'$ );
28 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 Alg. 10, line [3]).

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

- BinaryApplyAndReduce obtains two tuples, $inducedValueTuple$ and $inducedReductionTuple$, which describe the collapsing of duplicate leaf values, by calling the subroutine $CollapseClassesLeftmost$ (Alg. 9):
 - 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

Algorithm 14: InsertBConnection

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

```

1 begin
2   if there exists  $i \in [1..g.numberOfBConnections]$  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

```

values. For example, if deducedValueTuple is $[2, 2, 1, 1, 4, 1, 1]$, then inducedValueTuple is $[2, 1, 4]$.

The use of leftward folding is dictated by Structural Invariant 2b.

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

(See Alg. 10, line [4], as well as Alg. 9.)

- Finally, BinaryApplyAndReduce performs a corresponding reduction on Grouping g, by calling the subroutine Reduce, which creates a new Grouping in which g’s exit vertices are folded together with respect to tuple inducedReductionTuple (Alg. 10, line [5]).

Procedure Reduce, given as Alg. 13, recursively traverses Grouping g, working in the backwards direction, first processing each of g’s B-connections in turn, and then processing g’s A-connection. In both cases, the processing is similar to the (leftward) collapsing of duplicate leaf values that is carried out by BinaryApplyAndReduce:

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

In Alg. 13, line [12], the value of reductionTuple is used to create a tuple that indicates the equivalence classes of targets of return edges for the B-connection under consideration (in terms of the new exit vertices in the Grouping that will be created to replace g).

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

`Tuple inducedReductionTuple` is used to make a recursive call on `Reduce` to process the B -connection; `inducedReturnTuple` is used as the return tuple for the Grouping returned from that call. Note how the call on `InsertBConnection` (Alg. 14) in line [15] of `Reduce` enforces structural invariant 4 of Defn. 4.1.¹³

- As the B -connections are processed, `Reduce` uses the position information returned from `InsertBConnection` to build up the tuple `reductionTupleA` (Alg. 13, line [16]). This tuple indicates how to reduce the A -connection of g .
- Finally, via processing similar to what was done for each B -connection, two tuples are obtained that describe the collapsing that needs to be carried out on the exit vertices of the A -connection, and an additional call on `Reduce` is carried out. (See Alg. 13, lines [18]–[21].)

Function caching (§5.3) is performed for `Reduce`, with respect to both arguments g and `reductionTuple`.

§L shows that the time for a call to $\text{Reduce}(n, rt)$ 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 wish for this table to ever be polluted by non-well-formed proto-CFLOBDDs. Thus, there is a subtle point as to why the grouping g constructed during a call on `PairProduct` meets structural invariant 4—and hence why it is permissible to call `RepresentativeGrouping`(g) in line [38] of Alg. 12. We give this proof in Appendix §D.

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

Example 7.1. Fig. 11 illustrates how the CFLOBDD for $\lambda x_0, x_1.T$ is created from the “or” (\vee) of the CFLOBDDs for $\lambda x_0, x_1.x_0 \oplus x_1$ and $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$. Fig. 11c is the result of calling `PairProduct` on the CFLOBDDs for $\lambda x_0, x_1.x_0 \oplus x_1$ and $\lambda x_0, x_1.x_0 \Leftrightarrow x_1$. After \vee is applied to the values in each of the terminal-value pairs $[F, T]$ and $[T, F]$, we obtain a mock-CFLOBDD that has two exit vertices associated with terminal value T . To restore the structural invariants and create a CFLOBDD, the two exit vertices must be folded together, and a reduction performed on each of the two B -connections. In each case, `Reduce` is called with `reductionTuple` [1, 1]. Because these reductions result in the same B -connection proto-CFLOBDDs with identical return edges (Fig. 11e and Fig. 11f), which would be discovered by `InsertBConnection` (Alg. 14), it is necessary to fold together the two middle vertices and perform a reduction on the A -connection: `Reduce` is called with `reductionTuple` [1, 1]. This step produces the CFLOBDD for $\lambda x_0, x_1.T$ (Fig. 11g).

For another example that illustrates `Reduce`, see Ex. L.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

¹³In 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 $(\text{Grouping}, \text{ReturnTuple})$ pairs.

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 [26, §3.3].

Such an approach does not seem to be possible with CFLOBDDs because reduction is not obtained as a side-effect of hash-consing. The flow of control in Reduce (Alg. 13) follows the sequence of elements of a matched path backwards. Reduce makes recursive calls for the B-connection proto-CFLOBDDs and then a recursive call for the A-connection proto-CFLOBDD (rather than working bottom-up from level-0 groupings to level- k groupings, which would be the analogue of the bottom-up construction performed with BDDs.) Consequently, our CFLOBDD implementation maintains the weaker invariant that the Groupings that appear in the hash-consing tables are the heads of fully-fledged proto-CFLOBDDs, not mockproto-CFLOBDDs—i.e., structural invariants (1)–(4) of Defn. 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 mockproto-CFLOBDDs that violate the proto-CFLOBDD structural invariants.

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

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

7.4 Representing Matrices and Vectors using CFLOBDDs

Matrices and vectors are important data structures used in quantum simulation (§10.2.2). In this subsection and following subsections, we discuss how to represent Boolean or non-Boolean matrices and vectors using CFLOBDDs, and how to perform operations on such representations of matrices and vectors.

7.4.1 Matrix Representation. We represent square matrices using CFLOBDDs by having the Boolean variables correspond to bit positions in the row and column indices. That is, suppose that M is a $2^n \times 2^n$ matrix; M is represented using a CFLOBDD over $2n$ Boolean variables $\{x_0, x_1, \dots, x_{n-1}\} \cup \{y_0, y_1, \dots, y_{n-1}\}$, where the variables $\{x_0, x_1, \dots, x_{n-1}\}$ represent the successive bits of x —the first index into M —and the variables $\{y_0, y_1, \dots, y_{n-1}\}$ 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 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 $x_0, y_0, x_1, y_1, \dots, x_{n-1}, y_{n-1}$ —or the *reverse interleaved* ordering—i.e., the order is $y_{n-1}, x_{n-1}, y_{n-2}, x_{n-2}, \dots, y_0, x_0$.

One nice property of the interleaved-variable ordering is that, as we work through each pair of variables in an assignment, the matrix elements that remain “in play” represent a sub-block of the full matrix. For instance, suppose that we have a Boolean matrix whose entries are defined by the

¹⁴Matrices 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.

function $\lambda x_0 y_0 x_1 y_1. (x_0 \wedge y_0) \vee (x_1 \wedge y_1)$, as shown below:

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

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

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

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

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

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

There is an important, non-standard consequence of using a CFLOBDD to represent a matrix that very likely is not apparent from the discussion above, having to do with the sizes of subproblems in a divide-and-conquer algorithm. In fact, the same issue arises in designing a divide-and-conquer algorithm over any data structure represented via a CFLOBDD, as illustrated in Fig. 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 Fig. 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 $O(\sqrt{P})$.

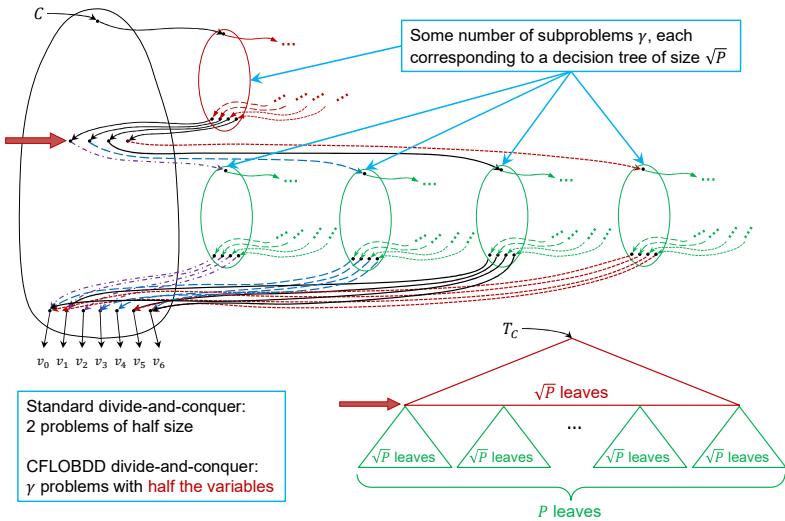


Fig. 12. Why a $\sqrt{P} \times \sqrt{P}$ -decomposition is the natural problem decomposition for divide-and-conquer algorithms on structures represented as CFLOBDDs.

Whereas with a decision tree it would be easy to take the conventional approach of dividing a problem into two problems of half *size*—using the left child and right child of the apex, in essence “peeling off” the topmost ply, such a division is not convenient for CFLOBDDs because the decision variable for the topmost ply is associated with the level-0 grouping found by following the A-connection of the A-connection of the . . . , etc. For CFLOBDDs, the natural structure of a divide-and-conquer algorithm lies with the A-connection proto-CFLOBDD and the set of B-connection proto-CFLOBDDs—a division based on dividing *the number of variables* in half.

In certain cases, including matrix multiplication (§7.7), the $\gamma \times \sqrt{P}$ -decomposition structure forced us to rethink how to perform various algorithms.

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

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

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

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

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

The CFLOBDD decomposition discussed above is different from (i) the natural decomposition of a matrix represented via a BDD, and (ii) the decomposition used in most divide-and-conquer algorithms on matrices. Both (i) and (ii) use $\frac{n}{2} \times \frac{n}{2}$ -decompositions (and thus decompose a matrix of size 16×16 into 4 sub-matrices, each of size 8×8 , and decompose a matrix of size 256×256 into 4 sub-matrices, each of size 128×128).

7.4.2 Vector Representation. A vector can be represented via a CFLOBDD in a manner that is similar to, but simpler than, the way matrices are represented. A vector of size $2^n \times 1$ can be represented by a CFLOBDD whose highest level is $\log n$. Suppose that V is a $2^n \times 1$ vector; a CFLOBDD representing V would have n Boolean variables $\{x_0, x_1, \dots, x_{n-1}\}$ with the variables $\{x_0, x_1, \dots, x_{n-1}\}$ representing the successive bits of x —the index into V .¹⁵

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

¹⁵Similar to matrices, vectors of other sizes can be represented using CFLOBDDs; for instance, they can be embedded within a larger vector whose dimensions are of the form $2^n \times 1$.

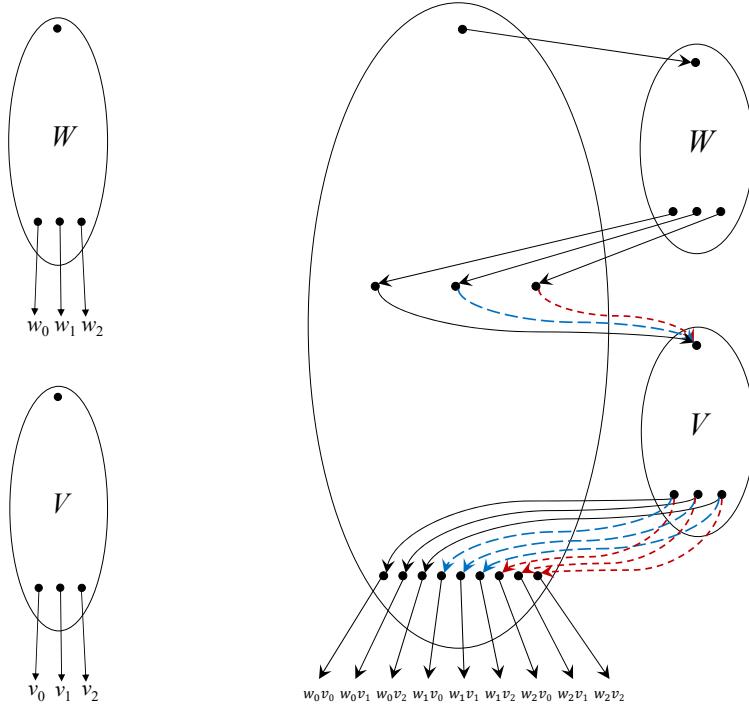


Fig. 13. (a) and (b) Level- k CFLOBDDs for arrays W and V , respectively; (c) level- $k + 1$ CFLOBDD for $W \otimes V$.

the vector would be as follows:

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

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

7.5 Kronecker Product

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

7.5.1 Variant 1. Suppose that matrices W and V are represented by level- k CFLOBDDs with value tuples $[w_0, \dots, w_m]$ and $[v_0, \dots, v_n]$, 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 $[w_0, \dots, w_m]$, and $(m + 1)(n + 1)$ exit vertices, corresponding to the terminal values

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

Algorithm 15: ShiftToAConnectionAtLevelOne

Input: Grouping g with variable ordering $x \bowtie y$

Output: Grouping g' is equal to g with dummy variables w' and z' such that the variable ordering is $((x \bowtie w') \bowtie (y \bowtie z'))$

```

1 begin
2   InternalGrouping  $g' = \text{new InternalGrouping}(g.\text{level} + 1);$ 
3   if  $g.\text{level} == 1$  then
4      $g'.\text{AConnection} = g;$ 
5      $g'.\text{AReturnTuple} = [1..g.\text{numberOfExits}];$ 
6      $g'.\text{numberOfBConnections} = |g'.\text{AReturnTuple}|;$ 
7     for  $i \leftarrow 1$  to  $g'.\text{numberOfBConnections}$  do
8        $g'.\text{BConnection}[i] = \text{NoDistinctProtoCFLOBDD}(g.\text{level});$ 
9        $g'.\text{BReturnTuples}[i] = [i];$ 
10    end
11     $g'.\text{numberOfExits} = g.\text{numberOfExits};$ 
12  else
13     $g'.\text{AConnection} = \text{ShiftToAConnectionAtLevelOne}(g.\text{AConnection});$ 
14     $g'.\text{AReturnTuple} = g.\text{AReturnTuple};$ 
15     $g'.\text{numberOfBConnections} = |g.\text{AReturnTuple}|;$ 
16    for  $i \leftarrow 1$  to  $g'.\text{numberOfBConnections}$  do
17       $g'.\text{BConnection}[i] = \text{ShiftToAConnectionAtLevelOne}(g.\text{BConnection}[i]);$ 
18       $g'.\text{BReturnTuples}[i] = g.\text{BReturnTuples}[i];$ 
19    end
20     $g'.\text{numberOfExits} = g.\text{numberOfExits};$ 
21  end
22  return RepresentativeGrouping( $g'$ );
23 end

```

where the terminal values are ordered lexicographically by their (i, j) indexes; i.e., $w_0v_0, w_0v_1, \dots, w_mv_{n-1}, w_mv_n$. 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_iv_j .
- (3) If any of the values in the sequence $[w_iv_j : i \in [0..m], j \in [0..n]]$ are duplicates, make an appropriate call on Reduce to fold together the classes of exit vertices that are associated with the same value, thereby creating a canonical multi-terminal CFLOBDD.

The construction through step (2) is illustrated in Fig. 13. Pseudo-code for the algorithm is presented in Appendix §G.

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

Algorithm 16: ShiftToBConnectionAtLevelOne

Input: Grouping g with variable ordering $w \bowtie z$

Output: Grouping g' is equal to g with dummy variables x' and y' such that the variable ordering is $((x' \bowtie w) \bowtie (y' \bowtie z))$

```

1 begin
2   InternalGrouping  $g' = \text{new InternalGrouping}(g.\text{level} + 1);$ 
3   if  $g.\text{level} == 1$  then
4      $g'.\text{AConnection} = \text{NoDistinctionProtoCFLOBDD}(g.\text{level});$ 
5      $g'.\text{AReturnTuple} = [1];$ 
6      $g'.\text{numberOfBConnections} = 1;$ 
7      $g'.\text{BConnection}[1] = \text{ShiftToBConnectionAtLevelOne}(g.\text{level});$ 
8      $g'.\text{BReturnTuples}[1] = g.\text{BReturnTuples}[1];$ 
9      $g'.\text{numberOfExits} = g.\text{numberOfExits};$ 
10  else
11     $g'.\text{AConnection} = \text{ShiftToAConnectionAtLevelOne}(g.\text{AConnection});$ 
12     $g'.\text{AReturnTuple} = g.\text{AReturnTuple};$ 
13     $g'.\text{numberOfBConnections} = |g.\text{AReturnTuple}|;$ 
14    for  $i \leftarrow 1$  to  $g'.\text{numberOfBConnections}$  do
15       $g'.\text{BConnection}[i] = \text{ShiftToAConnectionAtLevelOne}(g.\text{BConnection}[i]);$ 
16       $g'.\text{BReturnTuples}[i] = [1..g.\text{numberOfExits}];$ 
17    end
18     $g'.\text{numberOfExits} = g.\text{numberOfExits};$ 
19  end
20  return RepresentativeGrouping( $g'$ );
21 end

```

7.5.2 *Variant 2.* There is a second way to perform a Kronecker product of W and V that results in a representation of $W \otimes V$ that has the variable ordering $(x \bowtie w) \bowtie (y \bowtie z)$. (As discussed in §9.3.4, this version of Kronecker product is useful in Simon’s Algorithm.) The steps for $W \otimes V$ are as follows:

- For the CFLOBDD that represents matrix W , for every grouping of W from the top-level grouping down to level 2, create a copy of the grouping at one level greater. At level 1, create a level-2 grouping in which (i) the A-connection is the current level-1 grouping, and (ii) the B-connections are all the level-1 no-distinction proto-CFLOBDD (Fig. 7(b)). In essence, this step adds dummy variables that are proxies for matrix V ’s variables. Alg. 15 shows the algorithm for this operation.
- For the CFLOBDD that represents matrix V , for every grouping of V from the top-level grouping down to level 2, create a copy of the grouping at one level greater. At level 1, create a level-2 grouping in which (i) the A-connection is the level-1 no-distinction proto-CFLOBDD (Fig. 7(b)), and (ii) the B-connection is the current level-1 grouping. In essence, this step adds dummy variables that are proxies for matrix W ’s variables. Alg. 16 shows the algorithm for this operation.
- Finally, combine the two newly constructed CFLOBDDs by making a call to `BinaryApplyAndReduce` (Alg. 10), with “Times” (multiplication) as the operation to apply to terminal values.

Pseudo-code for this algorithm is given as Alg. 17.

Algorithm 17: Kronecker Product

Input: CFLOBDDs n_1, n_2 with variable ordering of $n_1: x \bowtie y$ and $n_2: w \bowtie z$
Output: CFLOBDD $n = n_1 \otimes n_2$ with variable ordering of $n: (x||w) \bowtie (y||z)$

```

1 begin
    // Add dummy variables  $\langle w'_1, \dots, w'_n \rangle, \langle z'_1, \dots, z'_n \rangle$  such that variable
    // ordering of  $g_1$  is  $(x \bowtie w') \bowtie (y \bowtie z')$ 
2   CFLOBDD g1 = RepresentativeCFLOBDD(ShiftToAConnectionAtLevelOne(n1.grouping),
    n1.valueTuple);
    // Add dummy variables  $\langle x'_1, \dots, x'_n \rangle, \langle y'_1, \dots, y'_n \rangle$  such that variable
    // ordering of  $g_2$  is  $(x' \bowtie w) \bowtie (y' \bowtie z)$ 
3   CFLOBDD g2 = RepresentativeCFLOBDD(ShiftToBConnectionAtLevelOne(n2.grouping),
    n2.valueTuple);
4   CFLOBDD n = BinaryApplyAndReduce(g1, g2, (op)Times);
5   return n;
6 end

```

7.6 Vector-to-Matrix Conversion

An important operation that is repeatedly performed in the algorithms summarized in §9.3 is vector-matrix multiplication. Our approach to vector-matrix multiplication is to convert a vector V of size $2^n \times 1$ into a matrix M of size $2^n \times 2^n$, where V occupies the first column, and all other entries of M are 0. We can then use the matrix-matrix multiplication algorithm discussed in §7.7.¹⁶ Note that the CFLOBDD representation of V has n variables and its highest level is $\log n$, whereas the CFLOBDD for matrix M has $2n$ variables and its highest level is $\log n + 1$.

We will denote the variables in the CFLOBDD representation of V as $x = \langle x_1, x_2, \dots, x_n \rangle$. The rows of M would use the same x variables. To represent the columns of M , we introduce an extra set of n variables: $y = \langle y_1, y_2, \dots, y_n \rangle$. As discussed in §7.4.1, we will use the interleaved ordering of x and y ($x \bowtie y$) to represent M ; i.e., the decisions (at level 0) by A-connection groupings at level 1 are the decisions for x variables, and the decisions (at level 0) by B-connection groupings at level 1 are those for y variables.

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

- (1) Create a new CFLOBDD c_1 of level $\log n + 2$ from V 's CFLOBDD representation c , by using level-0 groupings of c as the A-connection groupings of c_1 's level-1 groupings and adding *DontCareGroupings* as B-connection groupings at level 1 similar to Alg. 16. This step creates a CFLOBDD that represents a matrix in which every column is the vector V —because all of the column variables correspond to the added *DontCareGroupings*.
- (2) Create another CFLOBDD c_2 of level $\log n + 2$ that represents a matrix $Column1Matrix_n$ of size $2^n \times 2^n$ with only the first column filled with 1s and the rest with 0s.

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

¹⁶Matrix-vector multiplication is performed similarly.

Algorithm 18: Vector-to-Matrix Conversion

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

```

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

Column1Matrix_n can be recursively defined in terms of $\text{Column1Matrix}_{n/2}$ matrices of size $2^{n/2} \times 2^{n/2}$. This property allows c_2 to both be created and represented efficiently.

$$\text{Column1Matrix}_n = \begin{cases} \begin{bmatrix} \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \end{bmatrix}_{2^n \times 2^n} & n > 1 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & n = 1 \end{cases}$$

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

(3) Finally multiply c_1 and c_2 pointwise by calling `BinaryApplyAndReduce`.

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

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 . Steps 1 and 2 construct intermediate matrices M_1 and M_2 , defined as follows: $M_1 = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ and $M_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$. Finally, the result of converting vector V to a matrix is $M = M_1 * M_2 = \begin{bmatrix} 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, where “ $*$ ” denotes pointwise matrix multiplication.

7.7 Matrix Multiplication

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

The multiplication algorithm for CFLOBDDs presented here is similar to the standard $O(N^3)$ algorithm for multiplying two $N \times N$ matrices. There is a potential for savings because each of the argument CFLOBDDs may have a large number of shared substructures, and function caching can be

used to detect when a sub-problem has already been performed, in which case the proto-CFLOBDD for the answer can be returned immediately.

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

Suppose that P and Q are two $N \times N$ matrices represented by CFLOBDDs C_P and C_Q , respectively. The respective top-level A-connections, A_P and A_Q , are matrices of size $\sqrt{N} \times \sqrt{N}$ with matrix-valued cells of size $\sqrt{N} \times \sqrt{N}$. To multiply P and Q , we first recursively multiply A_P and A_Q . This operation defines which cells of A_P and A_Q get multiplied and added—and the answer is returned as a collection of symbolic expressions (of a form that will be described shortly). Using this information, we recursively call matrix multiplication and matrix addition on the B-connections, as appropriate. For the base case of the recursion—namely, level 1, which represents matrices of size 2×2 —we can enumerate all the individual cases of possible matrix structures (i.e., the patterns of which cells hold equal values), and build the CFLOBDDs that result from a matrix multiplication in each case.

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

Example 7.3. 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

$$\begin{bmatrix} ev_1 & ev_1 \\ ev_2 & ev_2 \end{bmatrix} \times \begin{bmatrix} ev'_1 & ev'_2 \\ ev'_1 & ev'_3 \end{bmatrix} = \begin{bmatrix} ev_1 ev'_1 + ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ ev_2 ev'_1 + ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix} = \begin{bmatrix} 2ev_1 ev'_1 & ev_1 ev'_2 + ev_1 ev'_3 \\ 2ev_2 ev'_1 & ev_2 ev'_2 + ev_2 ev'_3 \end{bmatrix} \quad (6)$$

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

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

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

$$\{\{(1, 1), 2\}\}, \{\{(1, 2), 1\}, \{(1, 3), 1\}\}, \{\{(2, 1), 2\}\}, \{\{(2, 2), 1\}, \{(2, 3), 1\}\}. \quad (7)$$

Now suppose that the two matrices are sub-matrices of level-2 groupings connected by ReturnTuples $rt = [5, 2]$ and $rt' = [6, 1, 2]$, respectively. Then applying $\langle rt, rt' \rangle$ to Eqn. (7) results in

$$\{\{(5, 6), 2\}\}, \{\{(5, 1), 1\}, \{(5, 2), 1\}\}, \{\{(2, 6), 2\}\}, \{\{(2, 1), 1\}, \{(2, 2), 1\}\}. \quad (8)$$

We call the objects shown in Eqns. (7) and (8) *MatMultTuples*. By this device, the answer to a matrix-multiplication sub-problem (whether from A-connections or B-connections, and at any level ≥ 1) can be treated as a multi-terminal CFLOBDD whose value tuple is a MatMultTuple.

Semantics of MatMultTuples. An alternative view of MatMultTuples comes from the right-hand matrix in Eqn. (6): a MatMultTuple is a sequence of bilinear polynomials over the exit vertices of two groupings. We will represent a bilinear polynomial p as a map from exit-vertex pairs to the corresponding coefficient. (The pairs for which the coefficient is nonzero are called the *support* of p . In examples, we show only map entries that are in the support.) In particular, suppose that g_1 and g_2 are two groupings at the same level, with exit-vertex sets EV and EV' . Each entry of a

Algorithm 19: Matrix Multiplication

Input: CFLOBDDs n_1, n_2
Output: CFLOBDD $n = n_1 \times n_2$

```

1 begin
2   Grouping×MatMultTuple [g,m] = MatrixMultOnGrouping(n1.grouping, n2.grouping);
3   ValueTuple v_tuple = [];
4   for  $i \leftarrow 1$  to  $|m|$  do
5     | Value  $v = \langle n_1.valueTuple, n_2.valueTuple \rangle(m(i))$ ;
6     | v_tuple = v_tuple || v;
7   end
8   Tuple×Tuple [inducedValueTuple, inducedReductionTuple] =
9     CollapseClassesLeftmost(v_tuple);
10  g = Reduce(g, inducedReductionTuple);
11  CFLOBDD n = RepresentativeCFLOBDD(g, inducedValueTuple);
12  return n;
13 end

```

MatMultTuple is of type $BP_{EV, EV'} \stackrel{\text{def}}{=} (EV \times EV') \rightarrow \mathbb{N}$. (We will drop the subscripts on BP if the exit-vertex sets are understood.)

To perform linear arithmetic on bilinear polynomials, we define

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

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 Eqn. (7) to Eqn. (8). For instance, let $rt = [1 \mapsto 5, 2 \mapsto 2]$ and $rt' = [1 \mapsto 6, 2 \mapsto 1, 3 \mapsto 2]$. Consider the second element of Eqn. (7): $bp = \{(1, 2), 1\}, \{(1, 3), 1\} = \{(1, 2) \mapsto 1, (1, 3) \mapsto 1\}$. Then the transformation of bp induced by rt and rt' can be expressed as follows (where Eqn. (9) expresses the general case):

$$\begin{aligned}
\langle rt, rt' \rangle(bp) &\stackrel{\text{def}}{=} \{(rt(ev), rt'(ev')) \mapsto bp(ev, ev') \mid ev \in EV, ev' \in EV'\} \tag{9} \\
&= \{(rt(1), rt'(2)) \mapsto bp(1, 2), (rt(1), rt'(3)) \mapsto bp(1, 3)\} \\
&= \{(5, 1) \mapsto 1, (5, 2) \mapsto 1\}
\end{aligned}$$

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

$$\langle vt, vt' \rangle(bp) \stackrel{\text{def}}{=} \sum \{bp(ev, ev') * vt(ev) * vt'(ev') \mid ev \in EV, ev' \in EV'\} \tag{10}$$

Algs. 19 and 20 give pseudo-code for the matrix-multiplication algorithm. Alg. 20 operates on two Groupings g_1 and g_2 at some level l , returning a (Grouping, MatMultTuple) pair (g, m) . Alg. 20 works symbolically, first considering the symbolic product of $g_1.AConnection$ and $g_2.AConnection$ (line [5]). The MatMultTuple ma returned from this call is really a list of directives: each directive

Algorithm 20: MatrixMultOnGrouping

Input: Groupings g_1, g_2
Output: Grouping \times MatMultTuple $[g, m]$ such that $g = g_1 \times g_2$

```

1 begin
2   if  $g_1.level == 1$  then                                // Base Case: matrices of size  $2 \times 2$ 
3     // Construct a level 1 Grouping that reflects which cells of the
4     // product hold equal entries in the output MatMultTuple
5   end
6   InternalGrouping  $g = \text{new InternalGrouping}(g_1.level);$ 
7   Grouping  $\times$  MatMultTuple  $[aa, ma] = \text{MatrixMultOnGrouping}(g_1.AConnection,$ 
8      $g_2.AConnection);$ 
9    $g.AConnection = aa; g.AReturnTuple = [1..|ma|]; g.numberOfBConnections = |ma|;$ 
10  // Interpret  $ma$  to (symbolically) multiply and add BConnections
11  MatMultTuple  $m = [];$ 
12  for  $i \leftarrow 1$  to  $|ma|$  do // Interpret  $i^{\text{th}}$  BP in  $ma$  to create  $g.BConnections[i]$ 
13    // Set  $g.BConnections[i]$  to the (symbolic) weighted dot product
14     $\sum_{((k_1, k_2), v) \in ma(i)} v * g_1.BConnections[k_1] * g_2.BConnections[k_2]$ 
15    CFLOBDD curr_cflobdd = ConstantCFLOBDD(g1.level, [0BP]);
16    for  $((k_1, k_2), v) \in ma(i)$  do
17      Grouping  $\times$  MatMultTuple  $[bb, mb] =$ 
18        MatrixMultOnGrouping(g1.BConnections[k1], g2.BConnections[k2]);
19      MatMultTuple mc = [];
20      for  $j \leftarrow 1$  to  $|mb|$  do
21        BP bp = ⟨g1.BReturnTuples[k1], g2.BReturnTuples[k2]⟩(mb(j));
22        mc = mc || bp;
23      end
24      Tuple  $\times$  Tuple  $[inducedMatMultTuple, inducedReductionTuple] =$ 
25        CollapseClassesLeftmost(mc);
26      bb = Reduce(bb, inducedReductionTuple);
27      CFLOBDD n = RepresentativeCFLOBDD(bb, inducedMatMultTuple);
28      curr_cflobdd = curr_cflobdd + v * n;           // Accumulate symbolic sum
29    end
30    g.BConnection[i] = curr_cflobdd.grouping;
31    g.BReturnTuples[i] = curr_cflobdd.valueTuple;
32    m = m || curr_cflobdd.valueTuple;
33  end
34  g.numberOfExits = |m|;
35  Tuple  $\times$  Tuple  $[inducedMatMultTuple, inducedReductionTuple] =$ 
36  CollapseClassesLeftmost(m);
37  g = Reduce(g, inducedReductionTuple);
38  return [RepresentativeGrouping(g), m];
39 end

```

is a bilinear polynomial used to create one of the BConnections of g . The workhorse computation—lines [9]–[24]—sets $g.BConnections[i]$ to the (symbolic) weighted dot product

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

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

Alg. 19 multiplies two matrices, `n1` and `n2`, represented as CFLOBDDs. It initiates the process at top level by calling `MatrixMultGrouping` on `n1.grouping` and `n2.grouping` (line [3]). It then uses the returned `MatMultTuple` as a list of directives, similar to the way `MatMultTuples` are used in `MatrixMultGrouping`. The difference is that at top level one has access to the actual values of the argument matrices, namely, `n1.valueTuple` and `n2.valueTuple`. Thus, in Alg. 19 the elements of `MatMultTuple m` are interpreted as directives to perform a *concrete* weighted dot product (lines [4]–[7]), using Eqn. (10) in line [5], thereby computing the values of the elements of the answer matrix

7.8 Path Counting and Sampling

A CFLOBDD whose terminal values are non-negative numbers can be used to represent a discrete distribution over the set of assignments to the Boolean variables. An assignment—or equivalently, the corresponding matched path in the CFLOBDD—is considered to be an elementary event. The “weight” of the elementary event is the terminal value. The probability of a matched path p is the weight of p divided by the total weight of the CFLOBDD—the sum of the weights obtained by following each of the CFLOBDD’s matched paths. Fortunately, it is possible to compute the aforementioned denominator by computing, for each of the terminal values, the number of matched paths that lead to that terminal value (§7.8.1). With those numbers in hand, it is then possible to sample an assignment/path according to the distribution that the CFLOBDD represents (§7.8.2).

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

7.8.1 Path Counting. Recall that every terminal value is connected to one exit vertex of the top-level grouping of the CFLOBDD. Every exit vertex of a grouping is, in turn, connected to exit vertices of internal groupings. Therefore, to compute the number of matched paths for every terminal value, we need to compute the path-counts from the entry vertex of a grouping to every exit vertex of that grouping, for every grouping in the CFLOBDD. For each grouping g , we would like to compute a vector of path-counts, in which the 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.

Algorithm 21: CountPaths

```

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

Alg. 21 shows pseudo-code for the path-counting algorithm. The path-counts are stored in a Tuple *numPathsToExit* in the Grouping data structure. Lines [13]–[20] perform the vector-matrix multiplication discussed above. In practice, because the number of path-counts increases double exponentially in the number of levels, we only store the log-values of the path-counts. It is also important for the implementation of the algorithm to perform function caching (§5.3) so that each grouping at each level is visited only once. When function caching is employed, Alg. 21 visits each grouping, and hence each vertex and edge of the CFLOBDD, exactly once; consequently, the cost of the path-counting operation is bounded by the size of the argument CFLOBDD.

This definition can also be stated equationally, in a form similar to the denotational semantics given in §6, where the expression in large brackets represents M_B .

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

Example 7.4. For the five proto-CFLOBDDs depicted in Fig. 9, the vectors of path-counts are computed as follows (read top-to-bottom by level):

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

7.8.2 Sampling an Assignment. Our goal is to sample a matched path from the distribution of matched paths of a given CFLOBDD, and return the corresponding assignment. As in §3.3.4, we assume that an assignment is an array of Booleans, whose entries—starting at index-position 1—are the values of successive variables. The concatenation of two such arrays a_1 and a_2 is denoted by $a_1 || a_2$.

To explain how to sample from the set of assignments, we argue in terms of the structure of the corresponding matched paths. If the distribution of matched paths was given as a vector of weights, $W = [w_1..w_{2^l}]$, as one would have in the corresponding decision tree, the probability of selecting the p^{th} matched path is given by

$$\text{Prob}(p) = \frac{w_p}{\sum_{i=1}^{2^l} w_i}. \quad (11)$$

In a CFLOBDD that represents a distribution, we do not have access to W directly. Suppose that $W' = [w'_1, \dots, w'_K]$ 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 §7.8.1 provides us with part of what is needed via *NumPathsToExit* of the top-level grouping.

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

Thus, while Eqn. (11) becomes

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

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

$$\text{Prob}'(\text{Path ends at terminal value } w'_t) = \frac{w'_t \times \text{numPathsToExit}[t]}{\sum_{j=1}^K w'_j \times \text{numPathsToExit}[j]} \quad (12)$$

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

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

Consider a grouping g and a given exit vertex i . For each middle vertex m of g , there is some number of matched paths—possibly 0—from the entry vertex of g that pass through m and eventually reach exit vertex i . Those numbers of matched paths, when divided by $|P_{g,i}|$, represent a distribution D_i on the set of g 's middle vertices. Consequently, the first step toward uniformly sampling a matched path from the set $P_{g,i}$ is to sample the index of a middle vertex of g according to distribution D_i . Call the result of that sampling step m_{index} . Thus, to sample a matched path from the entry vertex of g to exit vertex i , we (i) sample a middle vertex of g according to D_i to obtain m_{index} ; (ii) uniformly sample a matched path from $g.A\text{Connection}$ with respect to the exit vertex of $g.A\text{Connection}$ that returns to m_{index} ; (iii) uniformly sample a matched path from $g.B\text{Connections}[m_{\text{index}}]$ 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 $(g.B\text{ReturnTuples}[k])^{-1}[i]$ denote the exit vertex of $g.B\text{Connections}[k]$ that leads to i ; i.e., $\langle j, i \rangle \in g.B\text{ReturnTuples}[k] \Leftrightarrow (g.B\text{ReturnTuples}[k])^{-1}[i] = j$.

The path-counts for the number of matched paths of g 's B-connections (available via the vector NumPathsToExit for each of g 's B-connections, denoted by, e.g., $\text{numPathsToExit}_{g.B\text{Connections}[k]}$) 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, e.g., $\text{numPathsToExit}_{g.A\text{Connection}}[k]$, 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 $g.A\text{Connection}$, $g.B[k]$ denotes $g.B\text{Connections}[k]$, and $g.BRT$ denotes $g.B\text{ReturnTuples}$):

$$\text{Prob}(m_{\text{index}}) = \frac{\text{numPathsToExit}_{g.A}[m_{\text{index}}] \times \text{numPathsToExit}_{g.B[m_{\text{index}}]}[(g.BRT[m_{\text{index}}])^{-1}[i]]}{g.\text{numPathsToExit}[i]} \quad (13)$$

Example 7.5. Consider the CFLOBDD depicted in Fig. 9, and suppose that the goal is to sample a matched path that leads to terminal value T . From Ex. 7.4, we know that (i) the outermost grouping has 7 matched paths that lead to T , and (ii) NumPathsToExit is $[3, 1]$ and $[4]$ for the upper and lower level-1 groupings, respectively. Both of the outermost grouping's middle vertices have return

edges that lead to T ; thus, from Eqn. (13), we should sample the middle vertices with probabilities

$$\text{Prob}(m_{\text{index}} = 1) = \frac{[3,1][1] \times [3,1][2]}{7} = \frac{3 \times 1}{7} = \frac{3}{7} \quad \text{Prob}(m_{\text{index}} = 2) = \frac{[3,1][2] \times [4][1]}{7} = \frac{1 \times 4}{7} = \frac{4}{7}$$

Once m_{index} has been selected in accordance with Eqn. (13), we recursively sample a matched path—and its assignment a_A —from $g.A\text{Connection}$ with respect to exit vertex m_{index} (step (ii)). We also recursively sample a matched path—and its assignment a_B —from $g.B\text{Connection}[m_{\text{index}}]$ with respect to the exit vertex $(g.B\text{ReturnTuples}[m_{\text{index}}])^{-1}[i]$ 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.

Alg. 22 gives pseudo-code for the algorithm for sampling an assignment from a CFLOBDD.

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

8 RELATIONS EFFICIENTLY REPRESENTED BY CFLOBDDS

In this section, we prove that there exists an exponential separation between CFLOBDDs and BDDs. We establish this result using three relations that can be efficiently represented by CFLOBDDs. Note that we do not assume any specific variable ordering when discussing the sizes of BDDs for the functions used to prove the separation. We use node counts in BDDs, and vertex counts and edge counts in CFLOBDDs as a proxy for memory. (Recall from footnote 4 that we use the term “node” solely for BDDs, whereas “groupings” and “vertices” (depicted as the dots inside groupings) refer to CFLOBDDs.)

Remark. Recently, Zhi and Reps [82] obtained a characterization of relative sizes in the opposite direction (i.e., a bound on CFLOBDD size as a function of BDD size, for all BDDs). They showed that for every BDD of size n , there is a corresponding CFLOBDD, which uses the same variable ordering, of size $O(n^3)$.

8.1 The Equality Relation EQ_n

Definition 8.1. The equality relation $EQ_n : \{0, 1\}^{n/2} \times \{0, 1\}^{n/2} \rightarrow \{0, 1\}$ on variables $(x_0 \cdots x_{n/2-1})$ and $(y_0 \cdots y_{n/2-1})$ is the relation $EQ_n(X, Y) \stackrel{\text{def}}{=} \prod_{i=0}^{n/2-1} (x_i \Leftrightarrow y_i) = \prod_{i=0}^{n/2-1} (\bar{x}_i \bar{y}_i \vee x_i y_i)$.

THEOREM 8.2 (EXPONENTIAL SEPARATION FOR THE EQUALITY RELATION). *For $n = 2^l$, where $l \geq 1$, EQ_n can be represented by a CFLOBDD with $O(\log n)$ vertices and edges. In contrast, a BDD that represents EQ_n requires $\Omega(n)$ nodes.*

PROOF.

CFLOBDD Claim. We claim that with the interleaved-variable ordering $\langle x_0, y_0, \dots, x_{n/2-1}, y_{n/2-1} \rangle$, the CFLOBDD representation of EQ_n uses $O(\log n)$ groupings, each of constant size (and hence uses $O(\log n)$ vertices and edges in total). Diagrams that illustrate the CFLOBDD representation are shown in Fig. 14. For EQ_2 (i.e., $l = 1$), the representation is shown in Fig. 14a. This CFLOBDD has two groupings, a fork grouping at level 0 and a level-1 grouping. In total, it has eight vertices and eleven edges. Note that the “success” and “failure” exits at level 1 are the left and right exits, respectively.

Algorithm 22: Sample an Assignment from a CFLOBDD

```

1 Algorithm SampleAssignment(n)
2   Input: CFLOBDD n
3   Output: Assignment sampled from n according to n.valueTuple
4   begin
5     i  $\leftarrow$  Sample(n.valueTuple); // Sample terminal-value index i via Eqn. (12)
6     Assignment a = SampleOnGroupings(n.grouping, i);
7     return a;
8   end
9 end
10 SubRoutine SampleOnGroupings(g, i)
11   Input: Grouping g, Exit index i
12   Output: Assignment sampled from g, corresponding to one of the paths leading to exit i
13   begin
14     if g.level == 0 then
15       if g == DontCareGrouping then
16         return (random() % 2) ? "1" : "0";
17       else // g == ForkGrouping so i  $\in$  [1, 2]
18         return (i == 1) ? "0" : "1"
19       end
20     end
21     Tuple PathsLeadingToI = [];
22     for j  $\leftarrow$  1 to g.numberOfBConnections do // Build path-count tuple from
23       which to sample
24       if i  $\in$  g.BReturnTuples[j] then // if jth B-connection leads to i
25         PathsLeadingToI = PathsLeadingToI || (g.AConnection.numPathsToExit[j] *
26           g.BConnections[j].numPathsToExit[k]), where i = BReturnTuples[j](k);
27       end
28     end
29     end
30     m_index  $\leftarrow$  Sample(PathsLeadingToI); // Sample middle-vertex index m_index
31     Assignment a = SampleOnGroupings(g.AConnection, m_index) ||
32       SampleOnGroupings(g.BConnection[m_index], k), where i =
33       BReturnTuples[m_index](k);
34     return a;
35   end
36 end

```

For EQ_n with $n > 2$ (i.e., $l > 1$), the representation is defined inductively, following the pattern shown in Fig. 14b. For all i , $1 \leq i \leq l = \log n$, a level- $(i+1)$ grouping of the form shown at the outermost level of Fig. 14b has an A-connection and—from the leftmost middle vertex—a B-connection to the proto-CFLOBDD for $EQ_{2^{i-1}}$. The leftmost exit vertex of the proto-CFLOBDD is the “success exit” for testing equality on 2^{i-1} pairs of variables.

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

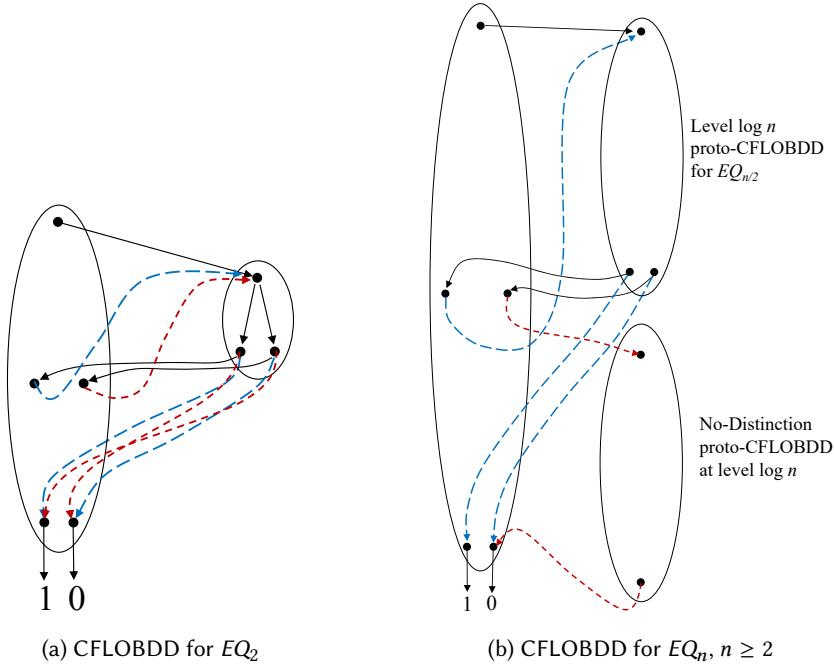


Fig. 14

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

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

The level- $(i+1)$ grouping has five vertices and eight edges, and the level- i grouping of the no-distinction proto-CFLOBDD at level i has three vertices and four edges (see Fig. 7d). Consequently, each of the $\log n + 1$ levels of the CFLOBDD for EQ_n contributes a constant number of vertices and edges, independent of i , and thus the total number of vertices and edges is $O(\log n)$.

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

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

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

- Case 2: Using an argument completely analogous to Case 1, we can show that none of the x variables can be dropped individually.
- Case 3: B depends on neither x_k nor y_k . Consider the following four assignments: $A_1 = \mathcal{T}$, $A_2 = \mathcal{T}[y_k \mapsto F]$, $A_3 = \mathcal{T}[x_k \mapsto F]$, and $A_4 = \mathcal{T}[x_k \mapsto F][y_k \mapsto F]$. Because B does not depend on either x_k or y_k , the function represented by B maps all four assignments to the same value (either 0 or 1), which violates the definition of the equality relation EQ_n (i.e., $EQ_n[A_1] = EQ_n[A_4] = 1$ and $EQ_n[A_2] = EQ_n[A_3] = 0$).

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

8.2 The Hadamard Relation H_n

The Hadamard Relation represents the family of Hadamard Matrices discussed in §2 and §3.4. The Hadamard matrices play a role in many quantum algorithms, including the seven that are used in §10.2.2 to evaluate the effectiveness of CFLOBDDs for simulating quantum circuits (namely, GHZ, BV, DJ, Simon’s algorithm, QFT, Shor’s algorithm, and Grover’s algorithm). See §9.3 and §10.2.2.

THEOREM 8.3 (EXPONENTIAL SEPARATION FOR THE HADAMARD RELATION). *The Hadamard Relation $H_n : \{0, 1\}^{n/2} \times \{0, 1\}^{n/2} \rightarrow \{1, -1\}$ between variable sets $(x_0 \cdots x_{n/2})$ and $(y_0 \cdots y_{n/2})$, where $n = 2^l$, can be represented by a CFLOBDD with $O(\log n)$ vertices and edges. In contrast, a BDD that represents H_n requires $\Omega(n)$ nodes.*

PROOF.

CFLOBDD Claim. As shown in §3.4, each matrix $H_n \in \mathcal{H}$, where $n = 2^l$ can be represented by a CFLOBDD with $O(l)$ vertices and edges—i.e., 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. The proof strategy follows a similar structure to the *BDD Claim* proof in Thm. 8.2. We prove the claim by contradiction. Suppose that there is some BDD B for H_n that does not need at least one node for each variable. In that case, the H_n function represented by B does not depend on that particular variable. Let \mathcal{T} denote the “all-true” assignment of variables; i.e., $\mathcal{T} \stackrel{\text{def}}{=} \forall k \in \{0..n/2-1\}, x_k \mapsto T, y_k \mapsto T$. There are three possible situations:

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

A_1 and A_2 yield the same value for the function represented by B , but they yield different values for the Hadamard relation. That is, if $H_n[A_1] = v$ (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 Fig. 2a, which is -1, and $H_2[A_2]$ is the value of the path $[x_0 \mapsto T, y_0 \mapsto F]$, which yields 1.
- $n = 4$. A_1 is the path to the rightmost (16^{th}) leaf in Fig. 2c, 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} , i.e., $H_{2^m}[A_1] = v$ (could be 1 or -1) and $H_{2^m}[A_2] = -v$. We must show that the claim holds true for $H_{2^{m+1}}$.

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

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

In both cases, the values obtained with a bit-flip do not match the value for an “all-true” assignment. \square

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

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

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

8.3 The Addition Relation ADD_n

Definition 8.4. The addition relation $ADD_n : \{0, 1\}^{n/3} \times \{0, 1\}^{n/3} \times \{0, 1\}^{n/3} \rightarrow \{0, 1\}$ on variables $(x_0 \cdots x_{n/3-1})$, $(y_0 \cdots y_{n/3-1})$, and $(z_0 \cdots z_{n/3-1})$ is the relation $ADD_n(X, Y, Z) \stackrel{\text{def}}{=} Z = (X + Y \bmod 2^{n/3})$.

THEOREM 8.5 (EXPONENTIAL SEPARATION FOR THE ADDITION RELATION). For $n = 3 \cdot 2^l$, where $l \geq 0$, ADD_n can be represented by a CFLOBDD with $O(\log n)$ vertices and edges. In contrast, a BDD that represents ADD_n requires $\Omega(n)$ nodes.

PROOF.

CFLOBDD Claim. For a given bit position i , the representation has to distinguish between two cases for the carry-bit value coming from bit position $i - 1$: *carry-in* = 0 and *carry-in* = 1. Fig. 15 shows decision trees for the *carry-in* = 0 and *carry-in* = 1 cases. The terminal values 0 and 1 indicate the carry-out value to be passed to bit position $i + 1$. Terminal value X represents a failure case: with the given carry-in value c_i , and the given values for x_i , y_i , and z_i , $z_i \neq c_i \oplus x_i \oplus y_i$.

The CFLOBDD representation for ADD_n using the interleaved-variable ordering is shown in Figs. 16–19. Our claim is that ADD_n has $O(\log n)$ vertices and edges. To keep triples of variables x_i , y_i , and z_i aligned with the power-of-two nature of CFLOBDDs, our representation of ADD_n uses an additional set of $n/3$ dummy variables (which will always be “routed” through a DontCareGrouping, so they have no effect on the relation over X , Y , and Z).

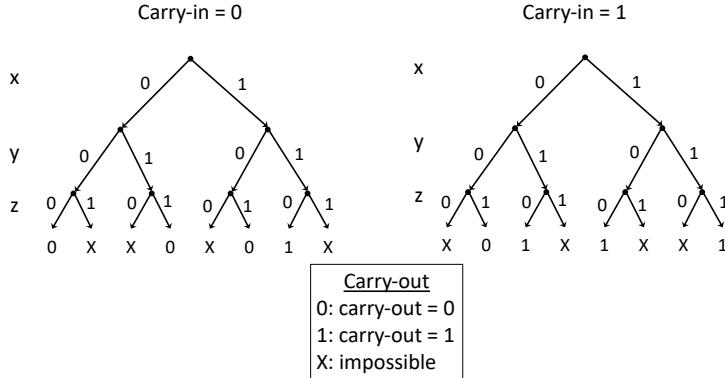


Fig. 15. Decision diagrams for the *carry-in* = 0 and *carry-in* = 1 cases of ADD_n

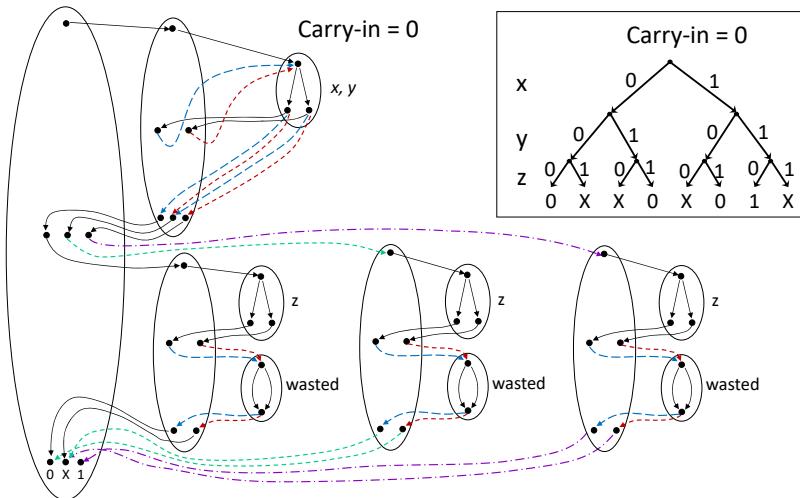


Fig. 16. CFLOBDD representation for the *carry-in* = 0 case of ADD_n

Fig. 16 shows the proto-CFLOBDD at level 2 for the *carry-in* = 0 case for some triple of variables x_i , y_i , and z_i in the ADD_n CFLOBDD. The DontCareGroupings labeled “wasted” in Fig. 16 are for the associated i^{th} dummy variable. Fig. 16 is one of two “leaf” building blocks that handles a triple of variables; the other is shown in Fig. 17, which depicts the CFLOBDD representation for the *carry-in* = 1 case for x_i , y_i , and z_i .

Fig. 18 shows the two recursive structures used at all higher levels (> 2) of the CFLOBDD for the *carry-in* = 0 and *carry-in* = 1 cases. Note that at all levels, including the base case of Fig. 16, the sequence for the exit vertices of *carry-in* = 0 proto-CFLOBDDs is $[0, X, 1]$. Similarly, at all levels the sequence for the exit vertices of *carry-in* = 1 proto-CFLOBDDs is $[X, 0, 1]$. Consequently, at each level, we can construct the *carry-in* = 0 and *carry-in* = 1 proto-CFLOBDDs by adding just one grouping for each case, and each of the groupings contains a fixed number of vertices and edges.

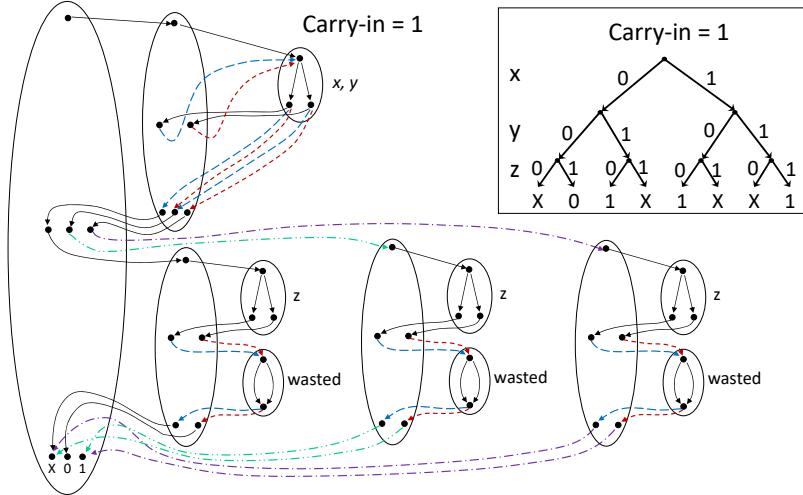


Fig. 17. CFLOBDD representation for the $carry-in = 1$ case of ADD_n

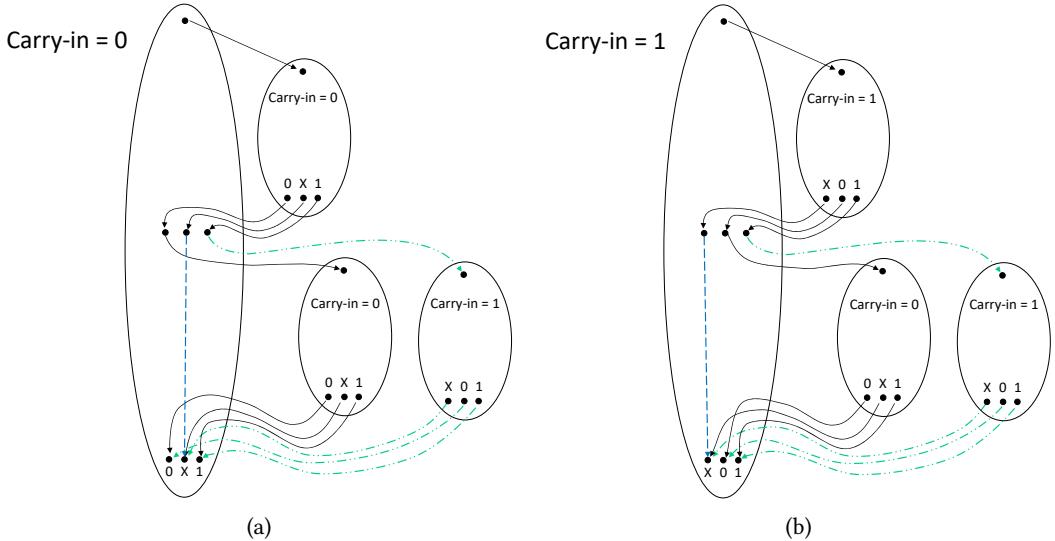


Fig. 18. Recursive CFLOBDD structures for the (a) $carry-in = 0$ and (b) $carry-in = 1$ cases of ADD_n . To reduce clutter, a B-connection to a NoDistinctionProtoCFLOBDD is depicted as a straight dashed line to the X exit vertex.

At the outermost level, we use the $carry-in = 0$ proto-CFLOBDD (which has an A-connection to the $carry-in = 0$ proto-CFLOBDD at the next lower level, but B-connections to both the $carry-in = 0$ and $carry-in = 1$ variants). The exit vertices labeled 0 and 1 are connected to terminal value T , and exit vertex X to terminal value F . Fig. 19a shows this structure.

To obey structural invariant 6 of Defn. 4.1, it is necessary to collapse the 0 and 1 exit vertices at the outermost level, because both lead to the terminal value T . This collapsing process propagates

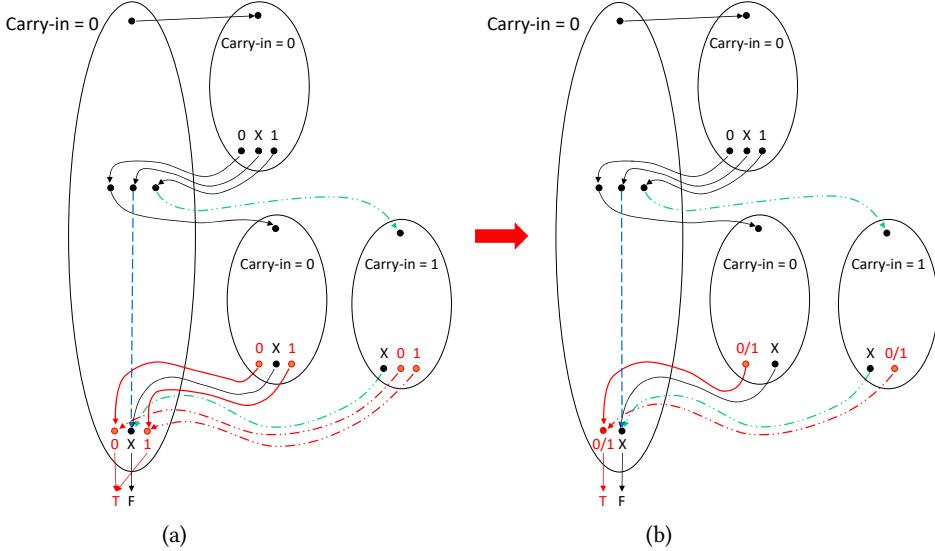


Fig. 19. (a) The recursive CFLOBDD structure of the ADD_n relation. Because the 0 and 1 exit vertices of the top-level grouping are associated with the single terminal T (while X is mapped to F), the 0 and 1 exits must be coalesced (indicated in red), which propagates to the internal groupings of the CFLOBDD. (b) The propagation of coalesced exit vertices.

to the different levels of internal groupings of the CFLOBDD. What remains to be established is that the collapsing step does not cause the CFLOBDD as a whole to blow up in size.¹⁷

Fortunately, as indicated in Fig. 19b, the propagation only takes place in groupings in B-connections (and B-connections of B-connections, etc.), and does not propagate to A-Connection groupings. As we prove below, the collapsing step only produces two more kinds of proto-CFLOBDDs at each level: (i) a *carry-in* = 0 variant in which the exit-vertex sequence is $[0/1, X]$, and (ii) a *carry-in* = 1 variant in which the exit-vertex sequence is $[X, 0/1]$ (where $0/1$ denotes the coalescing of the 0 and 1 exit vertices). Because there are now at most four grouping patterns that arise at each level, the final CFLOBDD at most doubles in size. The overall size of the CFLOBDD is proportional to the outermost level, and hence the CFLOBDD representation of ADD_n achieves double-exponential compression with respect to the size of the decision tree for ADD_n (i.e., $O(\log n)$ in vertices and edges, and $O(l)$ in the number of levels ($l = \log n$)).

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

$$R(l) = R(l - 1) + P(l - 1) + 7 + 13, \quad (14)$$

which defines $R(l)$ in terms of $R(l - 1)$ and $P(l - 1)$ according to the pattern given in Fig. 18a. Although Fig. 18a shows two *carry-in* = 0 proto-CFLOBDDs at level $l - 1$, they are actually shared

¹⁷The reason a “collapsing” step could cause a size blow-up is because occurrences of previously shared identical substructures could turn into multiple substructures, each slightly different.

data structures, and so $R(l - 1)$ only counts once in Eqn. (14). We call this property the “*same-structure sharing*” property: stated another way, all non-zero coefficients of R , P , A , and B terms can be replaced by 1. The 7 in Eqn. (14) represents the number of vertices at level l , and 13 refers to the edge count between groupings at level l and level $l - 1$. Similarly, the recurrence equation for $P(l)$, following the pattern in Fig. 18b, is

$$P(l) = P(l - 1) + R(l - 1) + 7 + 13. \quad (15)$$

Combining Eqns. (14) and (15), we obtain

$$\begin{aligned} R(l) + P(l) &= (R(l - 1) + P(l - 1) + 7 + 13) + (P(l - 1) + R(l - 1) + 7 + 13) \\ &= 2R(l - 1) + 2P(l - 1) + 2 * (7 + 13) && \text{Collecting terms} \\ &= R(l - 1) + P(l - 1) + 40 && \text{Same-structure sharing} \\ &= R(l - 2) + P(l - 2) + 2 * 40 && \text{Substitution} \\ &\vdots \\ &= R(l - k) + P(l - k) + k * 40 && \text{For a general } k \\ &\vdots \\ &= R(2) + P(2) + (l - 2) * 40 \\ &= O(1) + O(l) && \text{From Figs. 16 and 17} \\ &= O(l) \end{aligned} \quad (16)$$

Substituting Eqn. (Collecting terms) in Eqns. (14) and (15), we obtain

$$\begin{aligned} R(l) &= O(l - 1) + 20 \\ &= O(l - 1) + O(1) \\ &= O(l) \end{aligned} \quad (17)$$

$$\begin{aligned} P(l) &= O(l - 1) + 20 \\ &= O(l - 1) + O(1) \\ &= O(l) \end{aligned} \quad (18)$$

The argument for $A(l)$ and $B(l)$ is similar. We define $A(l)$ in terms of $R(l - 1)$, $A(l - 1)$, and $B(l - 1)$ following the pattern in Fig. 19b.

$$A(l) = R(l - 1) + A(l - 1) + B(l - 1) + 6 + 12, \quad (19)$$

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

$$B(l) = P(l - 1) + A(l - 1) + B(l - 1) + 6 + 12 \quad (20)$$

Combining Eqns. (19) and (20), we obtain

$$\begin{aligned} A(l) + B(l) &= (R(l-1) + A(l-1) + B(l-1) + 6 + 12) + (P(l-1) + A(l-1) + B(l-1) + 6 + 12) \\ &= R(l-1) + P(l-1) + 2A(l-1) + 2B(l-1) + 2 * (6 + 12) \end{aligned} \quad \text{Collecting terms}$$

$$= R(l-1) + P(l-1) + A(l-1) + B(l-1) + 36 \quad \text{Same-structure sharing}$$

$$= (R(l-2) + P(l-2) + 20) + (P(l-2) + R(l-2) + 20) \quad (21)$$

$$+ (R(l-2) + A(l-2) + B(l-2) + 6 + 12) \quad (22)$$

$$+ (P(l-2) + A(l-2) + B(l-2) + 6 + 12) + 36 \quad \text{Substitution}$$

$$= 3R(l-2) + 3P(l-2) + 2A(l-2) + 2B(l-2) + 40 + 36 + 36 \quad \text{Collecting terms}$$

$$= R(l-2) + P(l-2) + A(l-2) + B(l-2) + 40 + 2 * 36 \quad \text{Same-structure sharing}$$

⋮

$$= R(l-k) + P(l-k) + A(l-k) + B(l-k) + (k-1) * 40 + k * 36 \quad \text{For a general } k$$

⋮

$$= R(2) + P(2) + A(2) + B(2) + (l-3) * 40 + (l-2) * 36$$

$$= O(1) + O(l) \quad \text{From Figs. 16 and 17}$$

$$= O(l) \quad (23)$$

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

$$\begin{aligned} A(l) &= O(l-1) + O(l-1) + 18 \\ &= O(l-1) + O(1) \\ &= O(l) \end{aligned} \quad (24)$$

$$\begin{aligned} B(l) &= O(l-1) + O(l-1) + 18 \\ &= O(l-1) + O(1) \\ &= O(l) \end{aligned} \quad (25)$$

Eqns. (17), (18), (24), and (25) show that $R(l)$, $P(l)$, $A(l)$, and $B(l)$ are all linear in the number of levels— $O(l)$ —and logarithmic in the number of vertices and edges— $O(\log n)$. Thus, for the interleaved-variable ordering for (vectors of) variables X , Y , and Z , the vertices and edges count for the CFLOBDD for ADD_n is $O(\log n)$.

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

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

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

We have $ADD_n[A_1] = 1$ because the following triple of numbers is a correct instance of an addition problem:

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

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

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

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

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

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

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

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

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

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

9 APPLICATIONS TO QUANTUM ALGORITHMS

For certain problems, algorithms run on quantum computers achieve polynomial to exponential speed-ups over their classical counterparts. However, to date, there are no practical large-scale implementations of quantum computers. Hence, simulating quantum circuits on classical computers can provide insight on how quantum algorithms perform and scale. In this section and §10, we explore the potential of CFLOBDDs for simulating quantum circuits.¹⁸

¹⁸No knowledge of quantum algorithms is assumed. Everything can be understood in terms of some simple linear algebra. The only subtle concept is that some of the $2^n \times 2^n$ matrices are best thought of in terms of their effect on the *indices*

of positions in vectors of size $2^n \times 1$. For instance, the matrix $I \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 00 & 0 & 1 & 0 & 0 \\ 01 & 1 & 0 & 0 & 0 \\ 10 & 0 & 0 & 0 & 1 \\ 11 & 0 & 0 & 1 & 0 \end{bmatrix}$ maps the unit vectors

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

A *quantum circuit* takes as input an initial quantum-state vector, and applies a sequence of *quantum gates*, which are each length-preserving transformations, and can be expressed as unitary matrices. Thus, quantum-circuit simulation requires a way to perform linear algebra with vectors of size 2^n and matrices of size $2^n \times 2^n$, where n is the number of qubits involved. Examples of gates that operate on single qubits are $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, 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);¹⁹ 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 $\begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_0 \end{bmatrix}$. Let $M^{\otimes k}$ denote the k -fold

k occurrences of M

Kronecker product of M with itself: $M^{\otimes k} = \overbrace{M \otimes M \otimes \dots \otimes M}^k$. The quantum gate that, e.g., applies H to the j^{th} qubit of an n -qubit quantum state is $I^{\otimes(j-1)} \otimes H \otimes I^{\otimes(n-j)}$.

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

The evaluation of CFLOBDDs for quantum simulation in §10.2.2 uses the CFLOBDD representations of gate matrices and state vectors presented in this section, namely, multi-terminal CFLOBDDs with a semiring value at each terminal value (à la ADDs [7]). To support functions of type $\{0, 1\}^n \rightarrow \mathbb{C}$, we implemented a semiring of multi-precision-floating-point [28] complex numbers.

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

A completely different subscript convention is used to denote basis vectors: e_s denotes the vector with a one in position s (where s is interpreted in binary) and zeros elsewhere. To be concise, we sometimes use “exponent notation” from formal-language theory to express s . For instance, e_{0^n} and e_{1^n} denote the basis vectors $\underbrace{e_0 \dots 0}_{n \text{ copies}}$ and $\underbrace{e_1 \dots 1}_{n \text{ copies}}$, respectively.

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

Organization. In the present section, we articulate some advantages of quantum simulation (§9.1), and then discuss the application of CFLOBDDs to this domain. In §9.2, we give constructions of CFLOBDDs for some important families of vectors and matrices used in quantum algorithms. In §9.3, we briefly summarize some of the quantum algorithms used in the experiments in §10.2.2.

$e_{00} = \underbrace{\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}}_0$, $e_{01} = \underbrace{\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}}_1$, $e_{10} = \underbrace{\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}}_0$, and $e_{11} = \underbrace{\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}}_1$, and $e_{01} = \underbrace{\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}}_0$ to e_{01} , e_{00} , e_{11} , and e_{10} , respectively. In other words, on a unit vector,

the effect is to negate the final bit of the index that specifies the position of the 1.

¹⁹ A Hadamard gate that operates on a single qubit is the Hadamard matrix H_2 from Fig. 1 (§2), scaled by $\frac{1}{\sqrt{2}}$ so that it is a unitary matrix.

To aid the reader, the following table indicates where to find details about the CFLOBDD operations needed to simulate a quantum circuit:

<i>State construction</i>	Standard-basis vector	§9.2.4	Alg. 25
<i>Gate construction</i>			
	Identity gate	§9.2.2	Alg. 24
	Hadamard gate	Fig. 6b and §9.2.1	Alg. 23
	Not gate	§9.2.3	Variant of Alg. 24
	CNOT gate	§9.2.5	Appendix §I
<i>Operations</i>			
	Kronecker product	§7.5	Alg. 17
	Matrix-matrix multiplication	§7.7	Alg. 19
	Vector-matrix and matrix-vector multiplication	§7.6 and §7.7	Algs. 18 and 19
	Application of QFT	§9.2.6	Appendix §J
<i>Measurement</i>		§7.8	Alg. 22

9.1 Advantages of Simulation

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

- (1) A simulation can deviate from certain requirements of the quantum-computation model and perform the simulation in a way that no quantum device could.
 - (a) Some quantum algorithms perform multiple iterations of a particular quantum operator Op (e.g., k iterations, where k is some power of 2). *A simulation can operate on Op itself* [84, Ch. 6], using repeated squaring to create the sequence of derived operators $Op^2, Op^4, Op^8, \dots, Op^{2^{\log k}} = Op^k$, 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 (§9.3.6).
 - (b) The quantum-computation model requires the use of a limited repertoire of operations: every operation is a multiplication by a unitary matrix, and all results (and all intermediate values) must be produced in this way. In contrast, it is acceptable for a simulation to create some intermediate results by alternative pathways. 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 §9.2.5, which is used in §9.3.6.)
 - (c) In many quantum algorithms, the final state needs to be measured multiple times. When running on a physical quantum computer, part or all of the quantum state is destroyed after each measurement of the state, and thus the quantum steps must be re-performed before each successive measurement. In contrast, in a quantum simulation the quantum steps need only be performed once, and then *multiple measurements can be made because a (simulated) measurement does not cause any part of the simulated quantum state to be lost*.

Quantum supremacy refers to a computing problem and a problem size beyond which the problem can be solved efficiently on a quantum computer, but not on a classical computer. Quantum simulation is at one of the borders between classical computing and quantum computing: with quantum simulation, a classical computer performs the computation in roughly the same manner as a quantum computer, but can take advantage of shortcuts of

the kind listed above. In principle, a more efficient quantum-simulation technique has the potential to change the threshold for quantum supremacy.

- (2) Current quantum computers are error-prone and can lead to incorrect results, which is not the case with a simulation (modulo bugs in the implementation of the simulation).
- (3) Quantum simulation has a role in testing quantum computers. In particular, simulation can be used to create test suites for checking the correctness of the output states and measurements obtained from physical hardware.

9.2 Special Matrices

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

9.2.1 Hadamard Gate. As we saw in §3, a Hadamard matrix can be efficiently represented by a CFLOBDD. Hadamard matrices can be recursively defined as $H_{2^{l+1}} = H_{2^l} \otimes H_{2^l}$; thus, the Hadamard matrix at level $l + 1$ can be constructed using a Kronecker product of level- l Kronecker-product matrices (§7.5). Alternatively, it is possible to bypass such Kronecker-product operations and directly construct the Hadamard matrix for a given level. Pseudo-code for the latter approach is given as Alg. 23. The algorithm takes as input the max-level l , and returns the CFLOBDD of level l that represents the square matrix H_{2^l} of size $2^{2^{l-1}} \times 2^{2^{l-1}}$. The base case, for $l = 1$, returns the representation of

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

In Alg. 23, line [4], the value tuple is always $[1, -1]$ (and never $[-1, 1]$) because the $[0, 0]$ entry of every Hadamard matrix is 1.

As noted in footnote 19, a Hadamard gate is a Hadamard matrix scaled by a power of $\frac{1}{\sqrt{2}}$. In particular, the scale factor for H_{2^l} is $\left(\frac{1}{\sqrt{2}}\right)^l$. In our summaries of quantum algorithms in §9.3, we omit such scale factors to reduce clutter.

9.2.2 Identity Gate. The identity matrix, which is the same as the equality relation EQ_N , can be efficiently represented by a CFLOBDD, as shown in §8.1. The identity matrix at level $l + 1$ can be recursively computed from identity matrices at level l as $I_{2^{l+1}} = I_{2^l} \otimes I_{2^l}$. The CFLOBDD for the identity matrix at level $l + 1$ can either be constructed using a Kronecker product of level- l identity-matrix CFLOBDDs (§7.5), or it can be constructed directly. Pseudo-code for the latter approach is given as Alg. 24. The algorithm takes as input the max-level l , and returns the CFLOBDD of level l that represents I_{2^l} (of size $2^{2^{l-1}} \times 2^{2^{l-1}}$). The base case, for $l = 1$, returns the representation of

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

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

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

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

9.2.4 Standard-Basis Vectors. The family of standard-basis vectors are an important set of vectors in quantum algorithms because they represent the basis states. A standard-basis vector is a one-hot

Algorithm 23: Hadamard Matrix

```

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

```

vector—a vector all of whose elements are 0, except for a single 1. In the terminology of formal-language theory, the vector to picture is the *yield* of the decision tree obtained by unfolding the CFLOBDD (see Fig. 20). Let a standard-basis vector of size $2^n \times 1$ with its single occurrence of 1 at position *i* be denoted by e_x , where *x* is the binary representation of *i* using *n* bits. (The vector $e_0\dots$ is the initial state in many quantum algorithms.)

A representation as a CFLOBDD of a standard-basis vector for one-hot position *i* ($= x$) can be created in *n* steps via the pseudo-code given as Alg. 25. The code is relatively straightforward,

Algorithm 24: Identity Matrix

```

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

```

with the small complication that $x = 0 \dots 0$ is a special case that has to be accounted for at every level of recursion of auxiliary function StandardBasisVectorGrouping (as the bit-string x becomes of half-size, quarter-size, etc.—see lines [7]–[8] and lines [11]–[12]). The invariant for procedure StandardBasisVectorGrouping on input i ($= x$) is that exit vertex 2 always corresponds to e_x —unless $x = 0 \dots 0$, in which case exit vertex 1 corresponds to e_x .

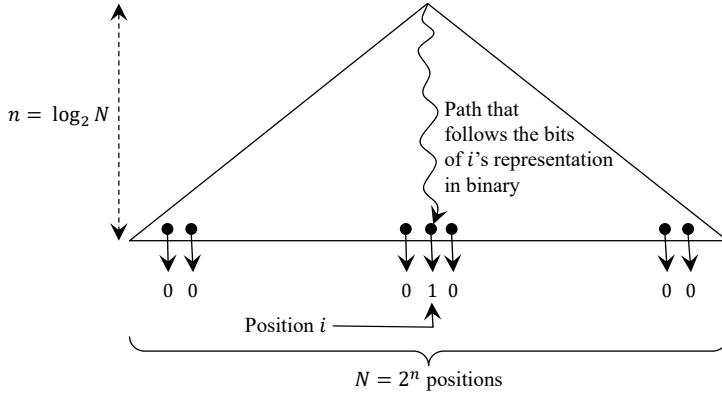


Fig. 20. One-hot vector as the yield of a decision tree. The single occurrence of 1 is at the leaf indexed by i —i.e., at the end of the path from the root that follows the bits of i 's representation in binary.

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

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

$$CNOT_2 = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 00 & 1 & 0 & 0 \\ 01 & 0 & 1 & 0 \\ 10 & 0 & 0 & 0 \\ 11 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} I_2 & O_2 \\ O_2 & X_2 \end{bmatrix} \quad (26)$$

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

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

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

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

Algorithm 25: Standard Basis Vector

1 **Algorithm** StandardBasisVectorCFLOBDD(l, i)
Input: int l - level of the CFLOBDD = $\log n$, where n = number of bits; int i - index
Output: The CFLOBDD that represents e_x where x = the binary representation of i in n bits

2 **begin**
3 Grouping g = StandardBasisVectorGrouping(l);
4 ValueTuple valueTuple = ($i == 0$) ? [1,0] : [0,1] // 1st elem. is 0 unless i is 0;
5 **return** RepresentativeCFLOBDD(g, valueTuple);
6 **end**
7 **end**

1 **SubRoutine** StandardBasisVectorGrouping(l, i)
Input: int l - level of the CFLOBDD = $\log n$, where n = number of bits; int i - index
Output: Grouping g that represents e_x where x = the binary representation of i in n bits.
(Exit vertex 2 corresponds to e_x unless $x = 0 \dots 0$, in which case exit vertex 1 corresponds to e_x .)

2 **begin**
3 **if** $l == 0$ **then**
4 **return** RepresentativeForkGrouping;
5 **end**
6 InternalGrouping g = new InternalGrouping(l);
7 int higherOrderIndex = $i \gg (1 \ll (l - 1))$ // First half of x ;
8 g.AConnection = StandardBasisVectorGrouping($l - 1, higherOrderIndex$);
9 g.ARearTuple = [1,2];
10 g.numberOfBConnections = 2;
11 int lowerOrderIndex = $i \& ((1 \ll (l - 1)) - 1)$ // Second half of x ;
12 Grouping g' = StandardBasisVectorGrouping($l - 1, lowerOrderIndex$);
13 **if** $higherOrderIndex == 0$ **then**
14 g.BConnection[1] = g' // Connection 1 = "on path" for x ;
15 g.BReturnTuples[1] = [1,2];
16 g.BConnection[2] = NoDistinctionProtoCFLOBDD($l - 1$);
17 g.BReturnTuples[2] = ($lowerOrderIndex == 0$) ? [2] : [1];
18 **else**
19 g.BConnection[1] = NoDistinctionProtoCFLOBDD($l - 1$);
20 g.BReturnTuples[1] = [1];
21 g.BConnection[2] = g' // Connection 2 = "on path" for x ;
22 g.BReturnTuples[2] = ($lowerOrderIndex == 0$) ? [2,1] : [1,2];
23 **end**
24 g.numberOfExits = 2;
25 **return** RepresentativeGrouping(g);
26 **end**
27 **end**

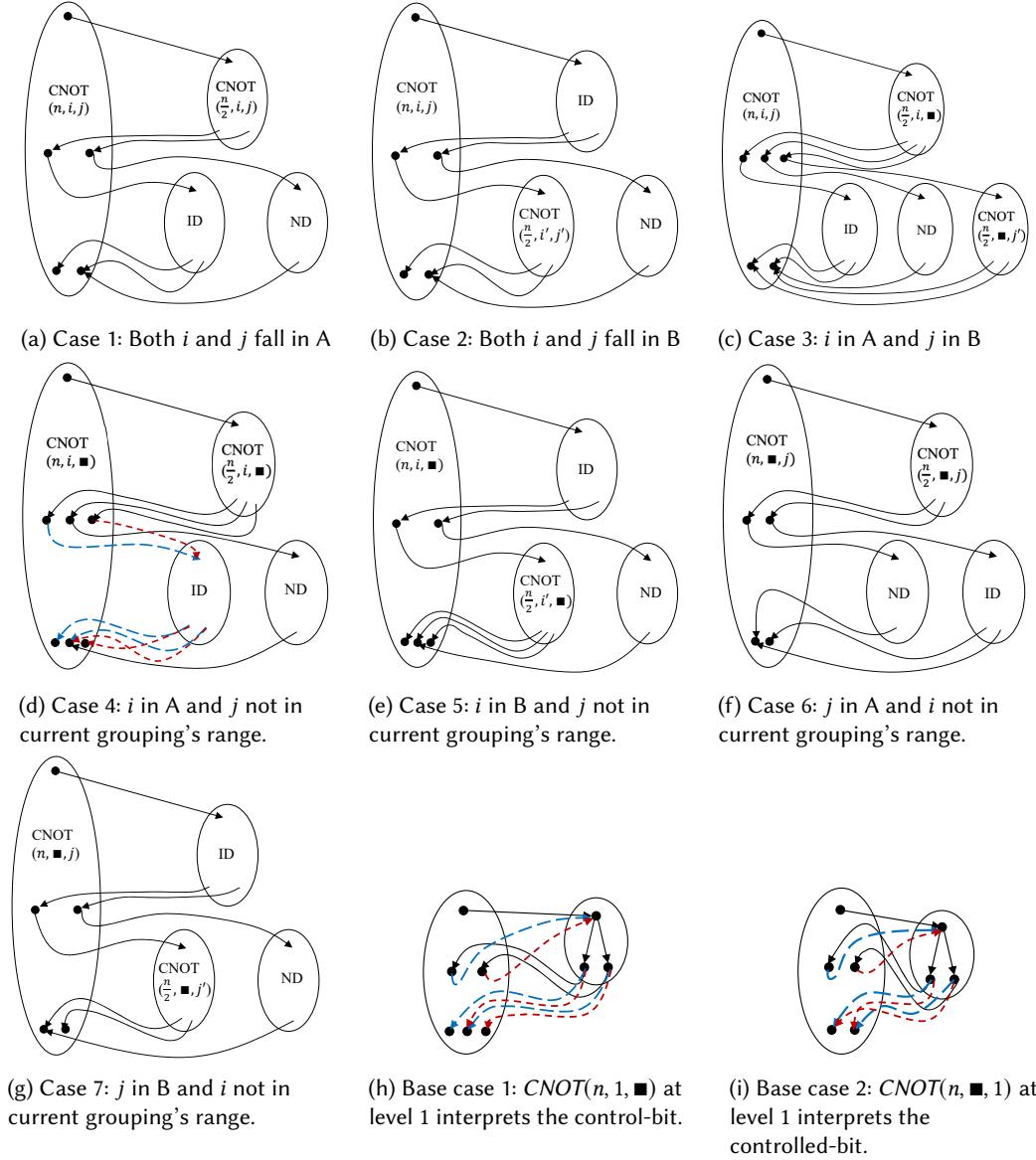


Fig. 21. The different cases of the CNOT construction. The text in each grouping denotes the function represented by the grouping. ID denotes IdentityMatrixGrouping; ND denotes a NoDistinctionProtocCFLOBDD (used here for an all-zero matrix). CNOT takes 3 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, and j for the controlled-bit, where $0 \leq i < j < n$. i' and j' denote bit indices adjusted according to the level l : $i' = i - 2^{l-1}$; $j' = j - 2^{l-1}$. A black square indicates that a particular index is outside the grouping's index range. Figures (h) and (i) show the two base cases at level 1, for $CNOT(n, 1, \blacksquare)$ and $CNOT(n, \blacksquare, 1)$, respectively.

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

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

Fortunately, we can create a double-exponentially compressed CFLOBDD representation of $CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$. Suppose that the control-bit is bit i , and the controlled-bit is bit j ; then for a grouping g , there are eight cases to consider (here we discuss the cases for $i < j$),²⁰ which are depicted in Fig. 21. The key to understanding Fig. 21 is that the construction maintains the invariant shown in Eqn. (27) on the exit vertices of the different kinds of groupings.

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

Here, “on-path” means that the exit occurs on a matched-path that can be continued to the top-level terminal value 1; “off-path” means that it will only be used to reach the top-level terminal value 0. For instance, in Fig. 21(a)–(g), each occurrence of ND (a NoDistinctionProtoCFLOBDD) is attached to the middle vertex of a grouping g that is reached from an A-connection’s off-path exit. The NoDistinctionProtoCFLOBDD has one exit vertex for which the return edge connects it to the off-path exit for g .

- (1) Both i and j fall in the A-connection of g . Fig. 21(a) shows the structural representation for this scenario. The bits in g ’s A-connection handle the CNOT operation. The bits in g ’s B-connection are not involved, and hence the “on-path” middle vertex is connected to Identity, and the “off-path” middle vertex is connected to a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix).
- (2) Both i and j fall in g ’s B-connection range. This scenario is depicted in Fig. 21(b). The bits in g ’s A-connection do not affect the CNOT operation, and hence we have an Identity matrix in the A-connection. The B-connection handles the relationship between the control-bit and controlled-bit through a CNOT structure of the form $CNOT(\frac{n}{2}, i', j')$. i' and j' denote bit indices adjusted according to the level l : $i' = i - 2^{l-1}$; $j' = j - 2^{l-1}$.
- (3) i lies in g ’s A-connection range and j in g ’s B-connection range. As shown in Fig. 21(c), g ’s A-connection handles the control-bit part, and hence g has 3 middle vertices. The first is the “on-path” case when the control-bit is 0, so no interpretation of the controlled-bit is needed; the second is the “off-path” case; and the third represents the information “the control-bit was activated.” Consequently, the B-connection groupings correspond to the respective three

²⁰The general case— $CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$, where j is allowed to be less than i —would be similar, but with some additional cases.

- cases: the first has the Identity matrix; the second a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix); and the third is a CNOT structure of the form $CNOT(\frac{n}{2}, \blacksquare, j')$.
- (4) i lies in g 's A-connection range, but j does not fall in g 's range (Fig. 21(d)). This case is similar to Fig. 21(c), in that g 's A-connection handles the control-bit part, and hence g has 3 middle vertices. The difference comes in the B-connections, which propagate the “states” of the middle vertices to g 's 3 exit vertices (using the Identity matrix for both “on-path” and “the control-bit was activated” and a NoDistinctionProtoCFLOBDD for “off-path”).
 - (5) i lies in g 's B-connection range, but j does not fall in g 's range. As shown in Fig. 21(e), the bits in g 's A-connection are not involved, and hence the A-connection is the Identity matrix. For the “on-path” middle vertex, g 's B-connection is to a CNOT structure of the form $CNOT(\frac{n}{2}, i', \blacksquare)$. For the “off-path” middle vertex, g 's B-connection is to a NoDistinctionProtoCFLOBDD.
 - (6) j lies in g 's A-connection range, but i does not fall in g 's range (Fig. 21(f)). g 's A-connection handles the part of the CNOT operation for flipping the controlled-bit and has 2 exit vertices. The bits of g 's B-connection are not involved; hence g 's “off-path” middle vertex is a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix) and the “on-path” middle vertex is the Identity matrix.
 - (7) j lies in g 's B-connection range, but i does not fall in g 's range (Fig. 21(g)). This case is similar to Fig. 21(f), except that the part of the CNOT operation for flipping the controlled-bit is displaced to the “on-path” B-connection (“on-path” from the Identity matrix in g 's A-connection). The “off-path” B-connection is a NoDistinctionProtoCFLOBDD (for an all-zero sub-matrix).
 - (8) There are two base cases of the recursive construction:

- Calls of the form $CNOT(n, 1, \blacksquare)$ at level 1 (Fig. 21(h)). The grouping g for this case represents a sub-matrix similar to the identity matrix I_2 , except that the second return edge of the second B-connection of g maps to a third (new) exit vertex of g (instead of mapping to the first exit vertex of g , as is the case for I_2). The third exit represents the information “the control-bit was activated” (i.e., the control-bit was 1), and hence, as indicated in Eqn. (27), for a matched-path to represent an element of the CNOT relation, there is an obligation to flip the value of the controlled-bit (elsewhere in the CFLOBDD).
- Calls of the form $CNOT(n, \blacksquare, 1)$ at level 1 (Fig. 21(i)). Note from Fig. 21(c) that the initial construction of the form $CNOT(n, \blacksquare, j)$ (at some higher level) is attached to the third middle vertex of the parent grouping, which, in turn, has a return edge from the third exit of the parent grouping's A-connection $CNOT(n, i, \blacksquare)$. Thus, $CNOT(n, \blacksquare, j)$ only occurs in a path context in which it is known that the control-bit was activated.

The $CNOT(n, \blacksquare, 1)$ grouping at level 1 interprets the controlled-bit. The proto-CFLOBDD used in this case is the same one found in both I_2 and X_2 . Inspection of Fig. 21(c), (f), and (g) reveals that such a proto-CFLOBDD is always connected to a level-2 grouping with return edges to middle or exit vertices that represent the “state-tuple” [off-path, on-path]. Consequently, each occurrence of a level-1 $CNOT(n, \blacksquare, 1)$ proto-CFLOBDD acts like the not matrix X_2 .

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

Complexity. Every CFLOBDD that represents a CNOT relation consists of only a constant number of kinds of groupings: as shown in Fig. 21, the representation of CNOT uses at most two kinds of CNOT groupings at level 1 and seven kinds of CNOT groupings at levels ≥ 2 , as well as proto-CFLOBDDs obtained from IdentityMatrixGrouping and NoDistinctionProtoCFLOBDD. Because at each level there are a constant number of groupings, each of constant size, the CFLOBDD

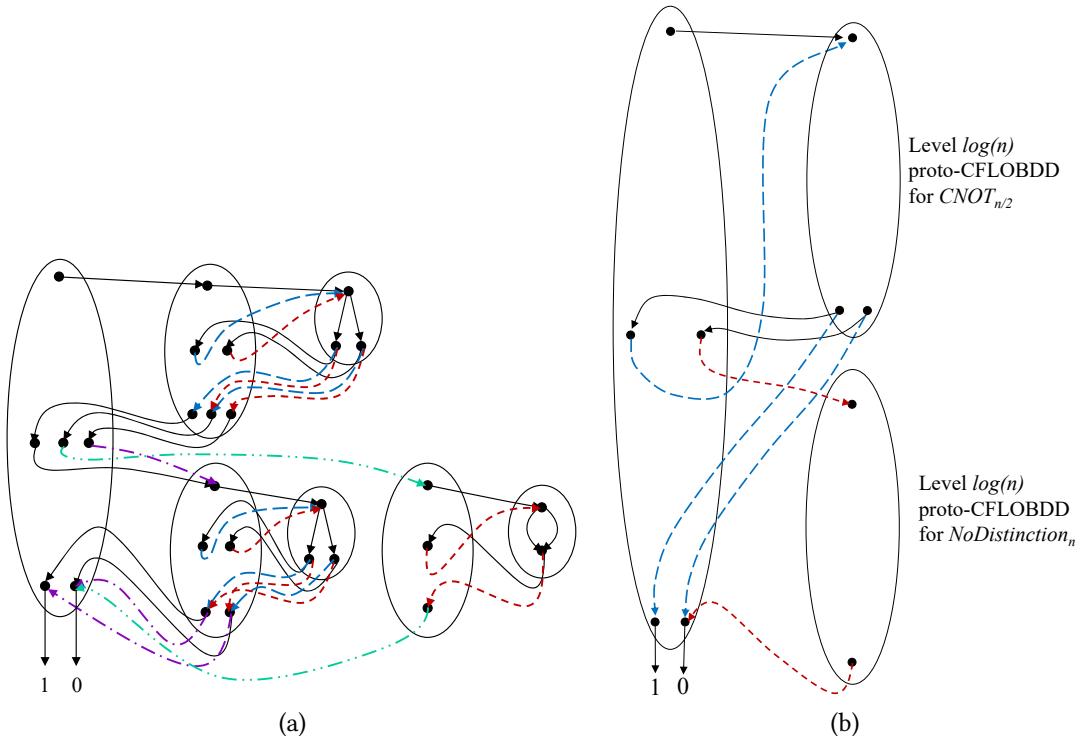


Fig. 22. CFLOBDD representation of $CNOT_n = CNOT_2^{\otimes n}$ with variable ordering $\langle x_1, y_1, x_2, y_2, \dots, x_n, y_n \rangle$. (a) Base case (level 2): $CNOT_2$ for 2 bits, with the first bit as the control-bit and second bit as the controlled-bit. (b) Grouping structure used for levels greater than 2.

representation of CNOT exhibits double-exponential compression compared to the size of the decision tree.²¹

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

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

$$CNOT_n = \prod_{i=2}^{2n} CNOT(2n, i, i+1). \quad (28)$$

²¹ The general case— $CNOT(n, i, j)$, $1 \leq i, j \leq n$, $i \neq j$, where j is allowed to be less than i —would still have at each level a constant number of kinds of groupings, each of constant size. Consequently, the general case also exhibits double-exponential compression compared to the size of the decision tree.

In Eqn. (28), each CNOT operation is between *adjacent* bits, and hence Eqn. (28) can be rewritten as

$$CNOT_n = \bigotimes_{i=1}^n CNOT_2 = CNOT_2^{\otimes n} \quad (29)$$

Moreover, we do not even have to perform the n explicit Kronecker-product operations of Eqn. (29) (or even $\log n$ Kronecker products) because $CNOT_n$ can be constructed directly (cf. item 1b in §9.1). Fig. 22 depicts the CFLOBDD representation of $CNOT_n = CNOT_2^{\otimes n}$, which has $\log n + 1$ levels and $2n$ Boolean variables, and the pseudo-code for the construction algorithm can be found in §I.

9.2.6 Quantum Fourier Transform. The Quantum Fourier Transform (QFT) is a linear transformation that is somewhat similar to the discrete Fourier transform. In matrix form, it looks like²²

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

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

We use the algorithm presented in [58] for the application of *QFT* to a state vector. More formally, we use the following steps to apply *QFT* for n qubit vector:

- (1) Start with a random vector e_x . (We use a randomly chosen basis vector as input in our experiments).
- (2) Apply the Hadamard matrix H_2 to the i^{th} qubit, where $i \rightarrow \{n \dots 1\}$.
- (3) For every i , apply a Controlled-Phase Gate CP from qubit $j \rightarrow \{1..i-1\}$ to i with a phase $= \frac{\pi}{2^{i-j}}$.
- (4) Finally, apply $n/2$ Swap Gates between i and $n - i$, where $i \rightarrow \{1..n/2\}$.

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

9.3 Quantum Algorithms

In this section, we briefly describe the quantum algorithms that are used in the experiments in §10.2.2. The ingredients of the algorithms are qubits, quantum gates, and measurements of qubits. Each of the states or operations can be viewed as algebraic operations on vectors and matrices. The mapping between the various aspects of the quantum algorithms and their corresponding algebraic operations is as follows:

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

We now discuss quantum algorithms as algebraic operations.

²²In the description of QFT, we vary from the subscript convention explained at the beginning of §9. Here, a gate matrix acting on n qubits (which is of size $2^n \times 2^n$) is denoted using a subscript N , where N denotes 2^n .

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

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

We extend the concept to n qubits by defining²³

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

We have used the algorithm given by Yu and Palsberg [81] for obtaining the GHZ state for n bits, which is as follows:

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

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

$$(H_{2n} \otimes H_2)(\prod_{i=1}^n CNOT(n+1, i, n+1))(H_{2n} \otimes X_2)(e_{0^{n+1}}) \quad (30)$$

Eqn. (30) is expressed in a way that uses vectors indexed by $n+1$ Boolean variables, and matrices indexed by $2n+2$ Boolean variables. Whereas a BDD implementation can directly encode Eqn. (30), CFLOBDDs are a representation of functions in which the number of Boolean variables must be a power of 2. For this reason, in a CFLOBDD implementation of GHZ, vectors and matrices are augmented so that vectors have $n-1$ dummy index variables and matrices have $2n-2$ dummy index variables. Taking into account these dummy variables, what is actually computed is the following:

$$(H_{2n} \otimes H_2 \otimes I^{\otimes n-1})(\prod_{i=1}^n CNOT(n+1, i, n+1))(H_{2n} \otimes X_2 \otimes I^{\otimes n-1})(e_{0^{2n}}) \quad (31)$$

By properties of Kronecker product,

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

Because the matrix-vector product $H_{2n} \times e_{0^n}$ results in a vector consisting of all ones, we can avoid performing an explicit multiplication and instead directly create a CFLOBDD whose top-level grouping is a `NoDistinctionProtoCFLOBDD` and whose terminal value is $\frac{1}{\sqrt{2^n}}$. In the implementation of these algorithms, a matrix-vector multiplication is implemented by first converting the vector $2^k \times 1$ to a matrix of size $2^k \times 2^k$ by padding with zeros (§7.6), and then performing matrix multiplication (§7.7).

In step (5), the terminal values (which represent amplitudes) are squared, and then the bit-strings of indices are sampled from the CFLOBDD based on the values of the squared amplitudes (which represent probabilities), as explained in §7.8.

²³Recall that we use e_{0^n} and e_{1^n} denote the standard-basis vectors $e_{\underbrace{0\dots 0}_{n \text{ copies}}}$ and $e_{\underbrace{1\dots 1}_{n \text{ copies}}}$, respectively.

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

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

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

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

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

$$(H_{2n} \otimes I_2)(U_{2(n+1)})(H_{2n} \otimes I_2)(e_{0^{n+1}}) \quad (32)$$

As in §9.3.1, to implement Eqn. (32) with CFLOBDDs, we introduce dummy index variables so that the total number of index variables is a power of 2. Also, in the term $(H_{2n} \otimes I_2)(e_{0^{n+1}}) = (H_{2n} \times e_{0^n}) \otimes (I_2 \times e_0)$, the multiplication $(H_{2n} \times e_{0^n})$ can be avoided by directly creating a CFLOBDD whose top-level grouping is a `NoDistinctionProtoCFLOBDD` and whose terminal value is $\frac{1}{\sqrt{2^n}}$.

9.3.3 Deutsch-Jozsa algorithm. The algorithm solves the following problem:

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

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

- (1) Prepare an initial state $e_{0^{n+1}}$ of $n+1$ bits (the standard-basis vector of $n+1$ bits with a 1 in the second position and 0s elsewhere).
- (2) Apply the Hadamard matrix $H_{2n} \otimes H_2$ on first $n+1$ bits.
- (3) Apply the oracle $U_{2(n+1)}$ to the current state.
- (4) Finally, apply the Hadamard matrix H_{2n} to the first n bits.
- (5) Measure the first n bits to obtain the string s . If $s = e_{0^n}$, then f is constant; otherwise, f is balanced.

Steps (1)–(4) can be expressed as:

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

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

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

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

- (1) Initialize the set of equations E to the empty set

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

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

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

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

Using Eqn. (28), and the interleaved Kronecker product (\otimes_i) discussed in §7.5.2 (Alg. 17), we can rewrite the above expression as follows:

$$(H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \quad (33)$$

Note that in Eqn. (33), we have changed to the bit ordering to $\langle x_1, y_1, x_2, y_2, \dots, x_n, y_n \rangle$.

Now consider the second, third, and fourth multiplicands in Eqn. (33):

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

Reading these terms right to left,

- first, the oracle U_{2n} is applied on the x bits, which yields $f(x)$;
- second, $CNOT_n$ copies the value in the x bits to the y bits, leading to the y bits also representing $f(x)$;
- third, U_{2n}^* is applied to the x bits, to turn the values in the x bits from $f(x)$ back to the original value of x .

However, we can achieve the same result by first copying the value of the x bits to the y bits by applying $CNOT_n$, and then applying U_{2n} directly on the y bits, which can be expressed as $(I_{2n} \otimes_i U_{2n})(CNOT_n)$. Thus, Eqn. (33) can be re-written as

$$(H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \quad (34)$$

Formally, to show that Eqns. (33) and (34) are equal, we use two properties of Kronecker product:

- (1) $A \otimes (B + C) = (A \otimes B) + (A \otimes C)$
- (2) $(A \otimes B)(C \otimes D) = (AC \otimes BD)$

Starting from Eqn. (33), we have

$$\begin{aligned}
 & (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n} \otimes_i I_{2n})(e_{0^n} \otimes_i e_{0^n}) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(H_{2n}e_{0^n} \otimes_i I_{2n}e_{0^n}) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(U_{2n} \otimes_i I_{2n})(\Sigma_x(e_x \otimes_i e_{0^n})) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(\Sigma_x(U_{2n}e_x \otimes_i I_{2n}e_{0^n})) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(CNOT_n)(\Sigma_x(e_{f(x)} \otimes_i e_{0^n})) \\
 &= (H_{2n} \otimes_i I_{2n})(U_{2n}^* \otimes_i I_{2n})(\Sigma_x(e_{f(x)} \otimes_i e_{f(x)})) \\
 &= (H_{2n} \otimes_i I_{2n})(\Sigma_x(U_{2n}^*e_{f(x)} \otimes_i I_{2n}e_{f(x)})) \\
 &= (H_{2n} \otimes_i I_{2n})(\Sigma_x(e_x \otimes_i e_{f(x)}))
 \end{aligned} \tag{33}$$

Starting from Eqn. (34), we have

$$\begin{aligned}
 & (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^{2n}}) \\
 &= (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n} \otimes_i I_{2n})(e_{0^n} \otimes_i e_{0^n}) \\
 &= (H_{2n} \otimes_i U_{2n})(CNOT_n)(H_{2n}e_{0^n} \otimes_i I_{2n}e_{0^n}) \\
 &= (H_{2n} \otimes_i U_{2n})(CNOT_n)(\Sigma_x(e_x \otimes_i e_{0^n})) \\
 &= (H_{2n} \otimes_i U_{2n})(\Sigma_x(e_x \otimes_i e_x)) \\
 &= (H_{2n} \otimes_i I_{2n})(I_{2n} \otimes_i U_{2n})(\Sigma_x(e_x \otimes_i e_x)) \\
 &= (H_{2n} \otimes_i I_{2n})(\Sigma_x(I_{2n}e_x \otimes_i U_{2n}e_x)) \\
 &= (H_{2n} \otimes_i I_{2n})(\Sigma_x(e_x \otimes_i e_{f(x)}))
 \end{aligned} \tag{34}$$

Eqns. (35) and (36) are identical, and hence Eqn. (34) can be used in place of Eqn. (33).

For step (11), our implementation uses the algorithm presented in [25, §2.1] for solving the set E of Boolean linear equations.

The implementation also illustrates two of the advantages of simulation discussed in §9.1:

- The discussion above about choosing to interleave the bits of x and y , and replacing $\prod_{i=1}^n (CNOT(2n, i, n+i))$ with $CNOT_n$ is in the spirit of item 1b.
- In accordance with item 1c, the implementation does not have to perform steps (2)–(7) for each sampling step. Instead, it performs steps (2)–(7) *once* to build up an appropriate CFLOBDD, and then samples the desired number of times from that structure.

9.3.5 Shor's Algorithm. Shor's Algorithm aims to find prime factors of a given integer N . We use the following algorithm:

- (1) Pick a random number $1 < a < N$.
- (2) Compute $K = \gcd(a, N)$. If $K \neq 1$, then K is a nontrivial factor of N , so we are done.
- (3) Otherwise, use the quantum period-finding algorithm (see below) to find r , which denotes the period of the function $f(x) = a^x \bmod N$. Equivalently, r is the smallest positive integer that satisfies $a^r \equiv 1 \bmod N$.
- (4) If r is odd, then go back to step (1).
- (5) If $a^{r/2} \equiv -1 \bmod N$, then go back to step (1).
- (6) Otherwise, both $\gcd(a^{r/2} + 1, N)$ and $\gcd(a^{r/2} - 1, N)$ are nontrivial factors of N , so we are done.

The quantum period-finding algorithm is as follows:

- (1) Start with $3n$ qubits with the first $2n$ qubits in state $e_{0^{2n}}$ and the last n qubits in state $e_{0^{n-1}}$.

- (2) For each qubit $i \rightarrow \{N..1\}$, apply the controlled gate $U_a^{2(N-i-1)}$ on the last n qubits, where $U_a(x) = ax \bmod N$.
- (3) Apply Inverse QFT on the first $2n$ qubits.
- (4) Measure the first $2n$ qubits to find r as discussed in [47, §11.6].

9.3.6 Grover's Algorithm. The algorithm solves the problem of finding a needle in a haystack. More formally,

Given a function $f : \{0, 1, \dots, N-1\} \rightarrow \{0, 1\}$, the algorithm finds x such that $f(x) = 1$.

Here, we assume that there is only one x such that $f(x) = 1$. The algorithm uses an oracle U_{2n} , which implements f for the “needle” w as

$$U_w(x) = \begin{cases} x & \text{if } x = w \\ -x & \text{otherwise} \end{cases}$$

Given this oracle, the algorithm for finding s is as follows:

- (1) Prepare an initial state e_{0^n} (the standard-basis vector of n bits with a 1 in the first position and 0s elsewhere).
- (2) Apply the Hadamard matrix H_{2n} on the initial state.
- (3) Apply the oracle U_w to the current state.
- (4) Apply the Grover diffusion operator $U_s = H_{2n}(2e_{0^n} \otimes_{outer} e_{0^n} - I_{2^n})H_{2n}$, where \otimes_{outer} denotes the outer product of two vectors.
- (5) Repeat steps (3)–(4) $\lceil \frac{\pi}{4} \sqrt{N} \rceil$ times.
- (6) Measure the n qubits to obtain the string s .

Steps (1)–(5) can be written concisely as follows:

$$\left(\prod_{i=1}^{\lceil \frac{\pi}{4} \sqrt{N} \rceil} U_s U_w \right) (H_{2n}(e_{0^n}))$$

As mentioned at the beginning of §9, one of the advantages of simulation is that we can re-associate the matrix multiplications in steps (1)–(4) to compute $\left(\prod_{i=1}^{\lceil \frac{\pi}{4} \sqrt{N} \rceil} U_s U_w \right)$ as an explicit quantity, which can be done using repeated squaring instead of as a sequence of multiplications. This approach provides a more efficient approach to steps (3)–(4). We can also use optimizations like those discussed earlier, e.g., performing (1)–(2) by means of a direct construction of the desired vector of all-1s. We can also create a representation of the diffusion operator U_s directly, rather than performing the computation given in step (4). In particular, the construction is almost the same as the identity-matrix construction in Alg. 24: one would use subroutine `IdentityMatrixGrouping`, but at top level the value tuple would be $[\frac{2}{N} - 1, \frac{2}{N}]$ (instead of $[1, 0]$).

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 CFLOBDDs²⁴ against a widely used BDD package, CUDD [71] (version 3.0.0), using CUDD’s C++ interface. The metrics are (i) execution time, and (ii) space (node counts in the case of BDDs; vertex counts + edge counts in the case of CFLOBDDs). We ran all experiments on AWS machines: t2.xlarge machines with 4 vCPUs, 16GB of memory, and a stack size of 8192KB, running on Ubuntu OS.

For RQ1 (§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) [24] (which can also be exponentially more succinct than BDDs), using Python package PySDD [50] (version 0.1).

For RQ2 (§10.2.2), we used a set of quantum-simulation benchmarks, and again compared the performance of CFLOBDDs against CUDD (version 3.0.0). For the quantum benchmarks, we did not enable dynamic variable reordering for BDDs because we could not retrieve the correct order of the output bits for a sampled string.

Five of the quantum benchmarks—BV, DJ, Simon’s algorithm, Shor’s algorithm, and Grover’s algorithm—use oracles for that either directly or indirectly incorporate the answer sought. Our methodology is standard for quantum-simulation experiments. Each benchmark uses a pre-processing step to create the CFLOBDD/BDD that represents the oracle. In each run, an answer is first generated randomly, and then the CFLOBDD/BDD that represents the oracle is constructed. Knowledge about the answer is used only during oracle construction. Thereafter, the quantum algorithm proper is simulated; these steps have no access to the pre-chosen answer (other than the ability to perform operations on the oracle, treated as a unitary matrix). The final step of running the benchmark is to check that the quantum algorithm obtained the correct answer.

We could not run the quantum benchmarks with SDDs because SDDs do not support multi-terminal values. However, we ran the quantum benchmarks using Quimb [32], a quantum-simulation library that uses tensor networks.

Extensions to CUDD. For the RQ2 experiments, we had to extend CUDD in two ways to be able to simulate quantum circuits using CUDD data structures:

- (1) CUDD supports *algebraic decision diagrams* (ADDs), which are multi-terminal BDDs with a value from a semiring at each terminal node. We had to extend CUDD with a semiring that was not part of the standard CUDD distribution (§10.1.1). For the corresponding experiments with CFLOBDDs, we used the same semiring for the terminal values of CFLOBDDs.
- (2) To allow quantum measurements to be carried out, we extended ADDs to support path sampling (i.e., selection of a path, where the probability of returning a given path is proportional to a function of the path’s terminal value).

10.1.1 Algebraic Decision Diagrams with Complex-Number Leaves. To use CUDD on most of the quantum benchmarks, we modified the ADD datatype and related functions to use multi-precision-floating-point complex numbers [28].

For the “Quantum Fourier Transform” and “Shor’s Algorithm” benchmarks, one only needs to represent a known set of complex roots of unity, so we used a different custom ADD datatype, *dtype*, defined as follows:

²⁴The implementation is available at <https://github.com/trishullab/cflobdd>.

```

typedef struct dtype {
    int val;
    int size;
    mpfr_t real;
    mpfr_t imag;
} dtype;
typedef dtype CUDD_VALUE_TYPE;

```

Here, “size” represents the number of roots of unity, and a value i for “val” represents the i^{th} root of unity with respect to the current “size.” For example, if size = 4, then val $\in \{0, 1, 2, 3\}$. Multiplying two *dtypes*— t_1 and t_2 with size = 4, where val of $t_1 = 2$ and val of $t_2 = 3$ —produces *dtype* t_3 with size = 4 and val = $(2 + 3) \% 4 = 1$. This representation encodes $\omega^2 \times \omega^3 = \omega$, where $\omega^k = e^{\frac{2\pi i}{4}}$ is a primitive 4th root of unity. We used this representation to compute the entire quantum Fourier transform, and then filled in the values for real and imag at the last step, where real = $\cos(\frac{2\pi * val}{size})$ and imag = $\sin(\frac{2\pi * val}{size})$.

10.1.2 Sampling in BDDs. An important step in quantum simulation is measurement of the output bits, which is equivalent to sampling a path from the BDD structure according to a probability distribution obtained from the terminal values. In particular, the terminal values represent so-called “amplitudes,” and the corresponding probability distribution is based on the squares of the amplitudes.

Path Counting. Suppose that the BDD has k terminals. To sample a path based on the squares of the amplitudes, we need to know, for each node n in the BDD, and each terminal position t_i , $1 \leq i \leq k$, the number of paths that lead from n to t_i . This information can be computed by generalizing the method of Ball and Larus for counting the number of paths in a DAG [8] from the case of a single exit-node to k exit-nodes (while also accounting for ply-skipping in a BDD):

- Each “path-count” is a k -dimensional vector c . (For convenience, we index the components of c as c_1, \dots, c_k .)
- Path-counts are computed bottom-up, starting from the terminal positions. The path-count at terminal position i has a 1 at index-position i and zeros elsewhere, signifying that (i) there is exactly 1 path from terminal node i to itself, and (ii) there are no paths from terminal node i to any of the other terminal nodes.
- When no plies are skipped in going from a node n to each of its two children, the path-count at n is the vector addition of the left-child and right-child path-counts.
- If Δ_{left} plies are skipped in going from n to the left child (with path-count c_l), and Δ_{right} plies are skipped in going to the right child (with path-count c_r), the path-count at n is $2^{\Delta_{\text{left}}}c_l + 2^{\Delta_{\text{right}}}c_r$.
- If Δ plies have been skipped at the apex of the DAG (with path-count c), the BDD’s overall path-count is $2^\Delta c$.

Sampling. Once path-counts are in hand, we sample a path as follows:

- Let $n.pc_i$ denote the path-count at a node n . Let $n.lchild$ and $n.rchild$ denote the two children of n , and let $n.\Delta_{\text{left}}$ and $n.\Delta_{\text{right}}$ denote the number of plies skipped in going to the left child and right child of n , respectively.
- For convenience, if Δ plies have been skipped at the apex a of the DAG, we assume that a new root node r is added whose left-child and right-child both point to a . $r.pc$ is set to $2^\Delta a.pc$, and both $r.\Delta_{\text{left}}$ and $r.\Delta_{\text{right}}$ are set to $\Delta - 1$. Otherwise, the root r is a .

- Let $p = \langle p_1, \dots, p_k \rangle$, denote the vector of squared amplitudes at the k terminal positions. The probability of choosing a path from r to terminal position i is $p_i * r.pc_i$.
- Randomly choose a value i based on the probabilities $\langle p_1 * r.pc_1, \dots, p_k * r.pc_k \rangle$. Henceforth, we only consult the i^{th} components of path-counts of the BDD's nodes.
- We work down the BDD from r to terminal position i , accumulating a path-string in π , which is initially set to ϵ . Starting at r , and then at each subsequently visited node n until terminal position i is reached, take the following steps:
 - If both $n.lchild.pc_i \neq 0$ and $n.rchild.pc_i \neq 0$, randomly choose one of the children in relative proportion to $2^{n.\Delta left} n.lchild.pc_i$ and $2^{n.\Delta right} n.rchild.pc_i$. If the left child is chosen, append “0” followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta left}$ to π ; otherwise, append “1” followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta right}$ to π .
 - If $n.lchild.pc_i = 0$, append “1” followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta right}$ to π .
 - If $n.rchild.pc_i = 0$, append “0” followed by a string chosen uniformly from $\{0, 1\}^{n.\Delta left}$ to π . Set n to the child chosen above.

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 $2^{n-1} \times 2^{n-1}$. (The aim of the benchmark is to test the performance of the matrix-multiplication and addition operations.)
- $ADD_n(X, Y, Z) \stackrel{\text{def}}{=} Z = (X + Y \bmod 2^{n/4})$, where X , Y , and Z are $n/4$ -bit integers.

Tab. 2 shows the performance of CFLOBDDs, BDDs (with and without dynamic reordering enabled), and SDDs within the 15-minute timeout threshold. For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1GB. For the ADD benchmark, BDDs (both with and without dynamic reordering) and SDDs ran out of memory within the 15-minute timeout threshold for problems with sufficiently many variables, even with such a large stack. (BDDs with dynamic reordering produced out-of-memory errors for $\#variables \geq 2^{24}$: the first step in the computation is to allocate the variables, which by itself leads to memory exhaustion for 2^{24} variables and beyond.) Note that, for SDDs, benchmark $MatMult_n$ is not applicable because SDDs do not handle non-Boolean values.

Because of a slightly technical alignment issue, our CFLOBDD representations of ADD_n deliberately waste one-quarter of the Boolean variables (as *dummy variables*). To make a fair comparison, our BDD and SDD encodings of ADD_n use only three-quarters of the Boolean variables indicated in column two of Tab. 2.

To understand how large a Boolean function could be created using CFLOBDDs (as a function of the number of Boolean variables),²⁵ we also measured the performance of the CFLOBDD implementation on the micro-benchmarks using a timeout of ninety minutes.

Fig. 23 shows graphs of size (#vertices + #edges) and time versus the number of Boolean variables for the three benchmarks.²⁶ Fig. 23a 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

²⁵The stack size was increased to 1GB for the runs with more than 2^{215} Boolean variables.

²⁶Tab. 2 shows the comparison of CFLOBDDs, BDDs, and SDDs for examples with a 15-minute timeout. In contrast, Fig. 23 shows the results of the stress test that we performed, where we gave the CFLOBDD implementation a 90-minute timeout.

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					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		
	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						
	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			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				
	2^{24}	115	826	941	0.003						
	2^{25}	120	862	982	0.003						
	:	:	:	:	:						
	2^{21}	10485755	75497434	85983189	113.99						
	2^{22}	20971515	150994906	171966421	385.75						
	2^{23}				Timeout (15min)						

Table 2. Performance of CFLOBDDs against BDDs, BDDs with dynamic reordering, and SDDs on the synthetic benchmarks for different numbers of Boolean variables. (For the two kinds of BDD experiments and the SDD experiments, we used a stack size of 1GB.)

up to $2^{22} = 4,194,304$ variables. Fig. 23b and Fig. 23c 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 $2^{23} = 8,388,608 \cong 4.27 \times 10^{2,525,222}$ variables, which comes to $0.75 \times 2^{8,388,608} \cong 3.12 \times 10^{2,525,222}$ after removing dummy variables.

Findings. CFLOBDDs performed better than BDDs and SDDs, both in terms of time and memory. For the benchmarks with more than 2^{18} Boolean variables, BDDs had memory issues. Using CFLOBDDs, it was also possible to construct representations of the benchmark functions with astounding numbers of Boolean variables: $2^{22} = 4,194,304$ for XOR_n ; $2^{27} = 134,217,728$ for $MatMult_n$; and $0.75 \times 2^{8,388,608} \cong 3.12 \times 10^{2,525,222}$ for ADD_n . These results support the claim that CFLOBDDs can provide substantially better compression of Boolean functions than BDDs.

10.2.2 RQ2: Do CFLOBDDs outperform BDDs when used for quantum simulation (in terms of time and space)? Tabs. 3 and 4 show the performance of CFLOBDDs and BDDs when simulating several well-known quantum algorithms, which are discussed in §9. In each case, for both CFLOBDDs and BDDs, we used the interleaved-variable ordering.

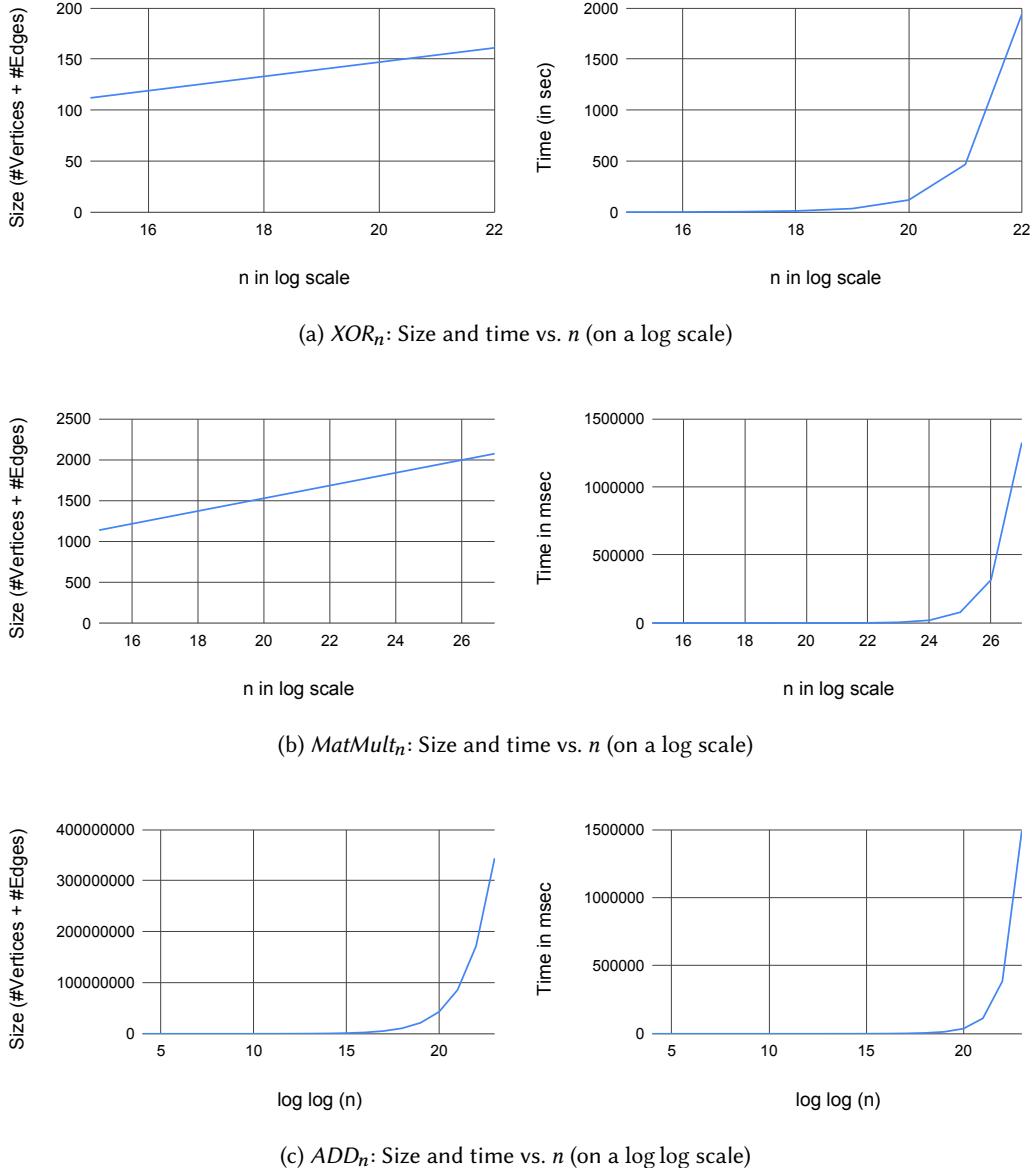


Fig. 23. CFLOBDD performance with a timeout of ninety minutes. Note that in (c) the number of Boolean variables is on a log log scale.

For GHZ, the algorithms do not depend on an input; the output is solely a function of the number of qubits used. For BV, DJ, QFT, Simon's algorithm, Shor's algorithm, and Grover's algorithm, we ran each algorithm with 50 different randomly selected inputs, for each of the indicated number of qubits. Tabs. 3 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

Benchmark	#Qubits	#Boolean Variables	CFLOBDD				BDD	
			#Vertices	#Edges	Total	Time (sec)	#Nodes	Time (sec)
GHZ	16	32	35	207	242	0.005	36	0.003
	32	64	43	255	298	0.007	68	0.008
	64	128	51	303	354	0.010	131	0.031
	128	256	59	351	410	0.015	259	0.143
	256	512	67	399	466	0.027	515	4.9
	512	1024	75	447	522	0.051	1028	44
	1024	2048	83	495	578	0.107	Timeout (15 min)	
	2048	4096	91	543	638	0.216		
	4096	8192	99	591	690	0.442		
	8192	16384	107	639	746	0.631		
	16384	32768	115	687	802	1.35		
	32768	65536	123	735	858	2.92		
	65536	131072	131	783	914	6.49		
	131072	262144	Timeout (15 min)					
BV	16	32	29	172	201	0.005	31	0.002
	32	64	39	233	272	0.006	63	0.004
	64	128	54	322	376	0.007	127	0.011
	128	256	76	456	532	0.010	255	0.040
	256	512	111	668	779	0.014	799	0.757
	512	1024	173	1039	1212	0.025	1027	39
	1024	2048	283	1701	1984	0.038	Timeout (15 min)	
	2048	4096	476	2854	3330	0.067		
	4096	8192	794	4762	5556	0.120		
	8192	16384	1337	8024	9361	0.335		
	16384	32768	2363	14177	16540	0.673		
	32768	65536	4391	26346	30737	1.42		
	65536	131072	8395	50372	58767	3.23		
	131072	262144	16220	97318	113538	8.46		
	262144	524288	31209	187251	218460	24.44		
	524288	1048576	58901	353404	412305	75.80		
	1048576	2097152	Timeout (15 min)					
DJ	16	32	18	90	108	0.006	18	0.001
	32	64	21	107	128	0.008	34	0.002
	64	128	24	123	147	0.008	66	0.038
	128	256	27	139	166	0.009	130	0.272
	256	512	30	154	184	0.01	258	2.1
	512	1024	33	170	203	0.011	516	795.5
	1024	2048	36	186	222	0.014	Timeout (15 min)	
	2048	4096	39	202	241	0.019		
	4096	8192	42	218	260	0.028		
	8192	16384	45	234	279	0.048		
	16384	32768	48	250	298	0.09		
	32768	65536	51	266	317	0.182		
	65536	131072	54	282	336	0.418		
	131072	262144	57	298	355	0.956		
	262144	524288	60	314	374	2.57		
	524288	1048576	63	330	393	7.8		
	1048576	2097152	66	346	412	26.15		
	2097152	4194304	69	362	431	95.57		
	4194304	8388608	72	378	450	180.33		
	8388608	16777216	Timeout (15 min)					

Table 3. 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 Alg.	16	64	583	16335	16918	0.71	5512	0.275
	32	128	123611	14096110	14219721	443.09	80243	3.31
	64	256	Timeout (90 min)				Timeout (90 min)	
QFT	4	8	7	73	80	0.001	31	0.0001
	8	16	9	572	581	0.034	255	0.001
	16	32	15	17868	17883	0.128	65535	0.098
	32	64	Timeout (15 min)				Timeout (15 min)	
Shor's Alg. (N, a) = (15, 2)	4	16	38	338	376	0.09	69	0.04
Shor's Alg. (N, a) = (21, 2)	5	16	72	877	949	2.13	136	0.72
Shor's Alg. (N, a) = (39, 2)	6	16	111	2443	2554	12.6	187	12.96
Shor's Alg. (N, a) = (69, 4)	7	16	176	4331	4487	53.47	605	30.38
Shor's Alg. (N, a) = (95, 8)	7	16	216	4928	5144	53.47	974	41.47
Shor's Alg. (N, a) = (119, 2)	7	16	220	7533	7753	53.47	3606	44.95
Shor's Alg. (N, a) = (323, 2)	9	32	Timeout (15min)				Timeout (15min)	
Grover's Alg.	16	32	17	91	108	0.009	47	0.214
	32	64	25	138	163	0.012	66	4.84
	64	128	38	212	250	0.018	Timeout (15 min)	
	128	256	58	333	391	0.030		
	256	512	91	531	622	0.080		
	512	1024	151	886	1037	0.292		
	1024	2048	259	1535	1794	14.11		
	2048	4096	450	2674	3124	64.85		
	4096	8192	766	4569	5335	909.86		
	8192	16384	Timeout (15 min)				Timeout (15 min)	

Table 4. Table (cont.) of the performance of CFLOBDDs against BDDs for increasing numbers of qubits.

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 §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 Tab. 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.”²⁷

In several cases, the problem sizes that completed successfully using CFLOBDDs were dramatically larger than the sizes that completed successfully using BDDs. With a 15-minute timeout, the number of qubits that CFLOBDDs can handle are 65,536 for GHZ, 524,288 for BV; 4,194,304 for DJ; and 4,096 for Grover's Algorithm, besting BDDs by factors of 128×, 1,024×, 8,192×, and 128×, respectively.

We also ran the CFLOBDD simulations with a 90-minute timeout, both to understand how execution time scales, as a function of number of qubits, and to see how large a problem instance can be handled. Fig. 24 shows the time taken (in seconds), with increasing numbers of qubits, for BV, GHZ, and DJ. With a 90-minute timeout, the BV and GHZ algorithms ran to completion with $2^{20} = 1,048,576$ qubits, and the DJ algorithm ran to completion with $2^{21} = 2,097,152$ qubits.

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

²⁷Given a , such that $1 < a < N$, the order-finding problem is to find the smallest positive integer r such that $a^r \equiv 1 \pmod{N}$.

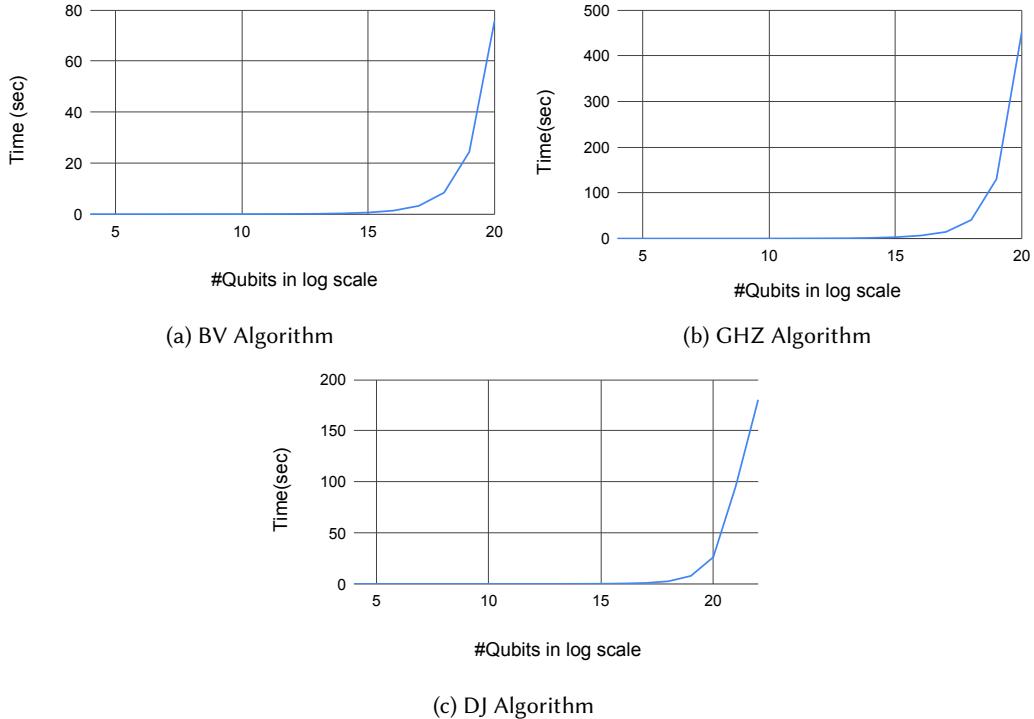


Fig. 24. Execution time (in seconds) vs. number of qubits (on a log scale) for three of the benchmarks.

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. In many of the algorithms, the initial and final CFLOBDD and BDD structures are of reasonable size, but there is an intermediate swell in size as the algorithm runs. Figs. 25 and 26 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). Fig. 26a shows that the CFLOBDD simulation of Grover's algorithm uses constant space from steps 3 to 15. The explanation is that, although the

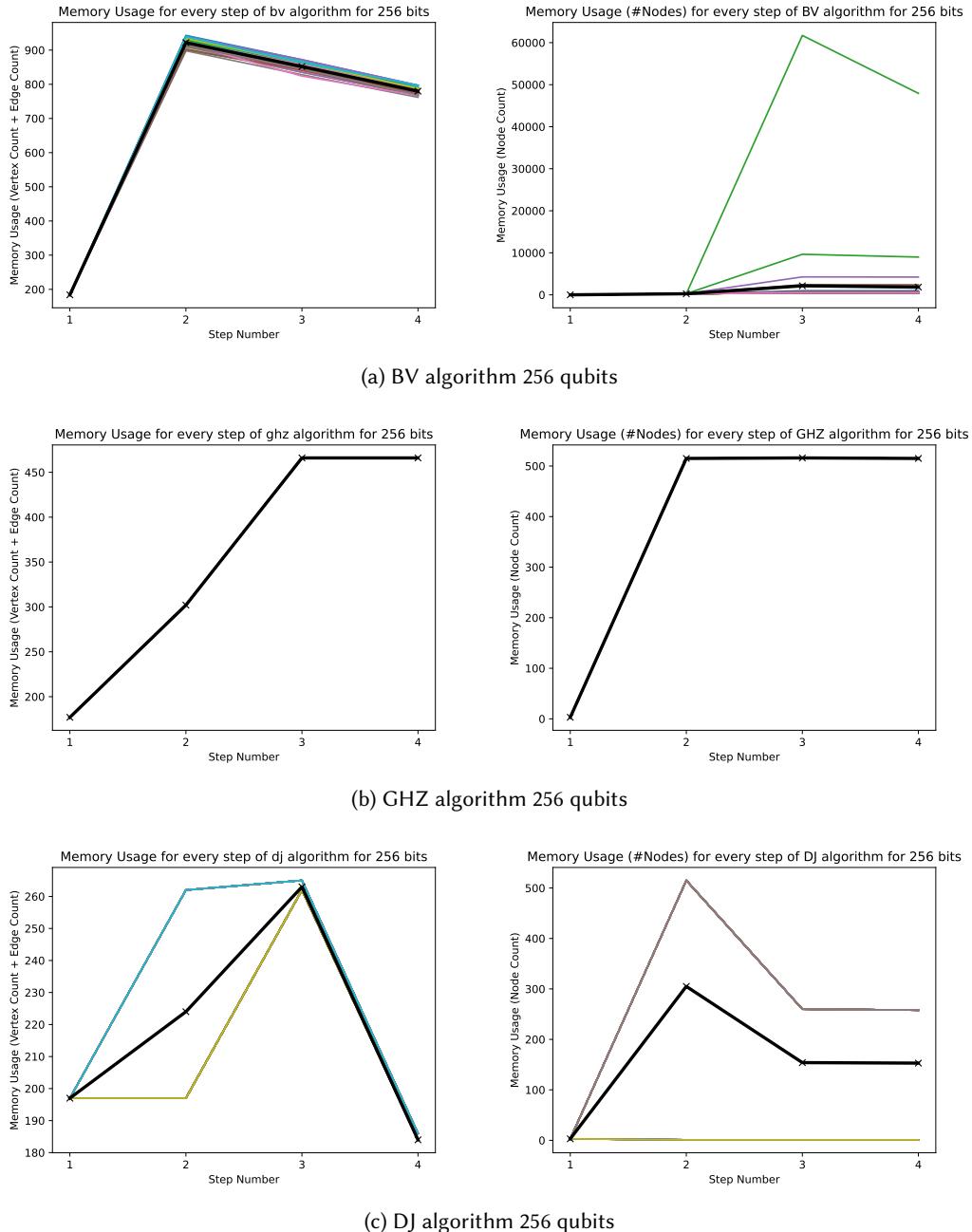


Fig. 25. Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.)

state vector changes at each step, the size of the CFLOBDD representation of the state vector does not change.

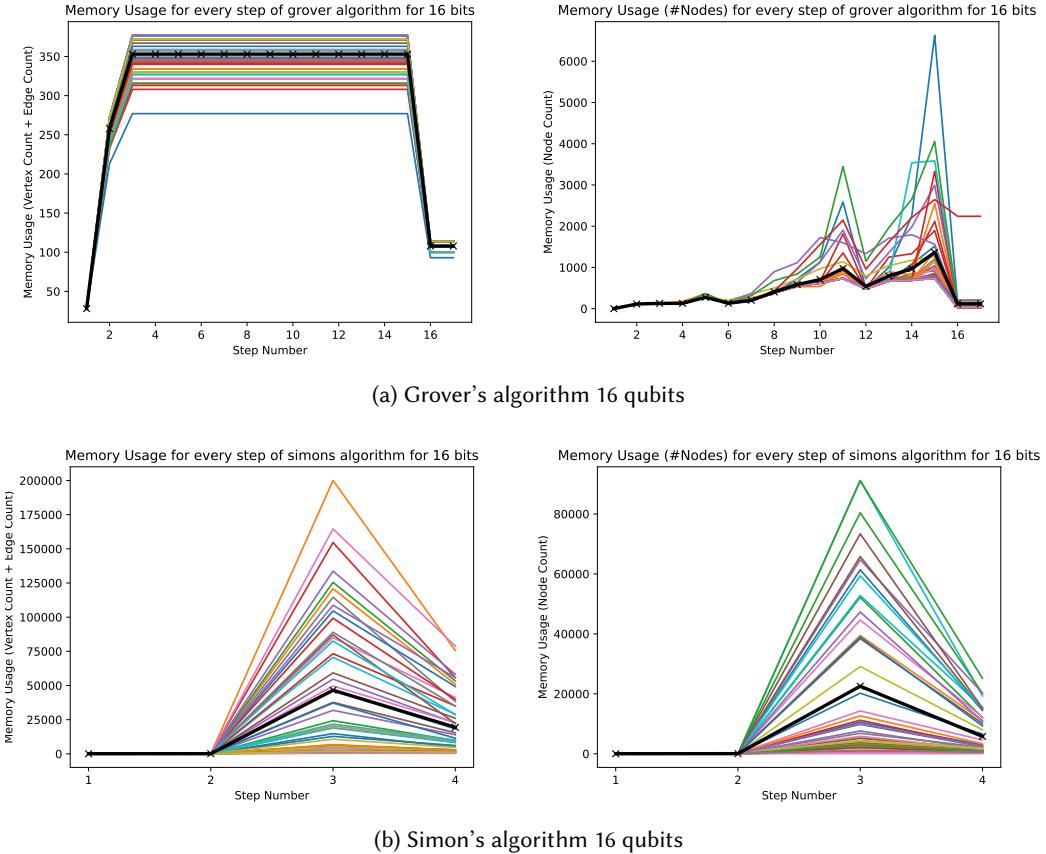


Fig. 26. Evolution of size through the steps of the indicated algorithms. (Left: CFLOBDD-based simulation; right: BDD-based simulation.)

Comparison with Tensor Networks. We also compared the performance of CFLOBDDs with Quimb [32], a state-of-the-art quantum simulator. Tab. 5 shows the performance of our CFLOBDD implementation and Quimb on the previously discussed quantum benchmarks. For the Quimb-based simulations of GHZ, BV, DJ, and Grover's algorithm, we used Matrix Product States (MPSs) [11, 74] and Matrix Product Operators (MPOs) [73] in algorithms modeled after the ones described in [79]. 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 [59] and the nearest-neighbor circuit mentioned in [29]. 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 [12], but the internal gates are created directly, as mentioned in [79] (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.²⁸

²⁸With 32 qubits, Quimb takes 1496.6sec \approx 25min.

These experiments show that, on some of the benchmarks, CFLOBDDs scale to much larger problem sizes than the Quimb tensor-network package, but on other benchmarks Quimb performs much better than CFLOBDDs.

What allows CFLOBDDs to perform so well on Grover’s algorithm? In each run of the CFLOBDD simulation of Grover’s algorithm, a random 4096-bit string s is chosen, then the Grover oracle matrix is constructed, along with the Grover diffusion operation, which are then multiplied together. A version of Grover’s algorithm based on repeated squaring of the product matrix is carried out (via operations that use the cumulative-product matrix—which depends on s —but the operations are 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 (§7.1.2), and (ii) the identity matrix (§9.2.2). In the largest cases of Grover’s algorithm that completed successfully within 15 minutes, the matrix has dimensions $2^{4096} \times 2^{4096}$; all off-diagonal entries are 0; and all diagonal entries are 1 except for the (s, s) entry, which is -1. To represent this matrix, one needs $8,192 = 2^{13}$ Boolean variables: 4,096 for the row-index and 4,096 for the column-index. There is a CFLOBDD representation of this matrix whose highest-level grouping is at level 13—thus, there are 14 levels in total, counting level 0. Moreover, the CFLOBDD has only a constant number of groupings at each of the 14 levels, so the matrix is one for which the CFLOBDD representation exhibits double-exponential compression.

Although multiplication of matrices represented by CFLOBDDs is not particularly efficient (see the last row of Tab. 1), there is little or no infill caused by the repeated-squaring operations, and so the matrix representation has only a limited amount of intermediate swell. (See the left-hand graph in Fig. 26(a) for a plot of memory usage for the CFLOBDD implementation of Grover’s algorithm for 16 qubits.)

11 RELATED WORK

Some History. CFLOBDDs were devised in the late 1990s; however, except for a rejected US patent application posted on the USPTO site in 2002 [66], nothing about them has ever been published. The idea and the preliminary implementation were set aside in about 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 (§8.3), and Boolean Haar wavelet transforms [39].

In late 2013, Reps read Lipton’s blog post from 2009, “BDD’s and factoring” [46]. In it, Lipton sketched a proposal for what he called *Pushdown BDDs*, a BDD-like structure based on nondeterministic pushdown automata. He speculated that even if the multiplication relation were of inherently exponential complexity for Pushdown BDDs, if the constant was small enough they could represent a threat to factoring-based cryptography. The realization that CFLOBDDs were closely related to a deterministic version of Pushdown BDDs, caused Reps to re-examine what relations could be expressed efficiently by a CFLOBDD, and to discover that ADD_n (§8.3) was such a relation.²⁹ Reps presented these results to Lipton in 2017, who suggested applying CFLOBDDs to quantum simulation. That effort was taken up by Sistla in summer 2018. This paper combines the heretofore unpublished material from the patent application with Sistla’s results on CFLOBDD-based quantum simulation.

²⁹As were a few others, such as Gray codes, with the interleaved-variable order.

Benchmark	#Qubits	CFLOBDD (Time in sec)	Quimb (Time in sec)
GHZ	16	0.005	0.222
	32	0.007	0.644
	64	0.010	2.29
	128	0.015	9.23
	256	0.027	40.31
	512	0.051	191.77
	1024	0.107	
	:	:	Timeout (15 min)
	65536	6.49	
	131072	Timeout (15 min)	
BV	16	0.005	0.264
	32	0.006	0.773
	64	0.007	2.75
	128	0.010	11.08
	256	0.014	49.49
	512	0.025	243.69
	1024	0.038	
	:	:	Timeout (15 min)
	524288	75.80	
	1048576	Timeout (15 min)	
DJ	16	0.006	0.256
	32	0.008	0.761
	64	0.008	2.75
	128	0.009	11.18
	256	0.010	49.33
	512	0.011	243.01
	1024	0.014	
	:	:	Timeout (15 min)
	4194304	180.33	
	8388608	Timeout (15 min)	
Simon's Alg.	16	0.71	2.56
	32	443.09	17.34
	64	Timeout (15 min)	267
	128	Timeout (15 min)	Timeout (15min)
QFT	4	0.001	0.023
	8	0.034	0.035
	16	0.128	0.074
	32		0.231
	64		1.64
	128		10.32
	256		103.65
	512		Timeout (15min)
	Shor's Alg. $(N, a) = (15, 2)$	4	0.08
	Shor's Alg. $(N, a) = (21, 2)$	5	0.1
Grover's Alg.	Shor's Alg. $(N, a) = (39, 2)$	6	0.11
	Shor's Alg. $(N, a) = (69, 4)$	7	0.12
	Shor's Alg. $(N, a) = (95, 8)$	7	0.12
	Shor's Alg. $(N, a) = (119, 2)$	7	0.12
	Shor's Alg. $(N, a) = (323, 2)$	9	0.27
	:	:	Timeout (15 min)
	Shor's Alg. $(N, a) = (6085, 8)$	12	107.28
	Shor's Alg. $(N, a) = (11611, 2)$	13	Out of Memory
		16	0.009
		32	0.012
		:	Timeout (15 min)
		4096	909.86
		8192	Timeout (15 min)

Table 5. Performance of CFLOBDDs against Quimb on quantum benchmarks for different numbers of qubits.

In the best case, a CFLOBDD for a function f can be double-exponentially smaller than the decision tree for f . There is a sense in which ROBDDs are incapable of such a degree of compression. *Quasi-reduced BDDs* are the version of BDDs in which variable ordering is respected, but don't-care nodes are *not* removed (i.e., plies are not skipped), and thus all paths from the root to a leaf 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 [75, Thm. 3.2.3]. Thus, although ROBDDs can give better-than-exponential compression compared to decision trees, what one has is not double-exponential compression: at best, it is linear compression of exponential compression. Moreover, in §8, we showed that the CFLOBDD for a function g can be exponentially smaller than any ROBDD for g .

Pre-History: Interprocedural Path Profiling. The idea behind CFLOBDDs was inspired by an obstacle encountered in interprocedural path profiling. In path profiling, a program is instrumented with code that counts the number of times particular path fragments of the program are executed. Ball and Larus [8] devised a clever technique that allowed *intraprocedural* path profiling to be carried out with low run-time overhead. Melski and Reps [53] generalized their technique to support profiling of path fragments that cross procedure boundaries. The Melski-Reps scheme performs an interprocedural analog of a transformation used in the Ball-Larus scheme:

- The Ball-Larus cycle-elimination transformation is performed on the CFG of each procedure.
- The interprocedural CFG is further transformed to “short-circuit” paths through recursive call sites, in effect, cutting each cycle that is present because of recursive procedure calls.

Interestingly, these two transformations do not leave one with an acyclic interprocedural CFG; the graph still contains cyclic paths. For example, via different call-sites, a path can exit a procedure and re-enter it from a distinct call-site. However—and this is the crucial fact on which this approach to interprocedural path profiling rests—there are only a *finite* number of “realizable paths,” in which each exit-to-return-site edge taken to a return site matches a preceding call-to-start edge from the corresponding call site. (Only the realizable paths represent “observable path fragments” that are to be reported during an execution run.)

The drawback of the Melski-Reps scheme is that there can be an enormous number of realizable paths in the transformed interprocedural CFG. In fact, the number of realizable paths can be *doubly exponential* in the size of the interprocedural CFG. For path profiling this situation is terrible: it means that the edge numbers that the instrumentation code has to manipulate can be enormous. For instance, Melski [51] found that one 20,000-line program had $2^{400,000}$ such paths—which would require the path-profiling instrumentation code for the program to manipulate 400,000-bit numbers! Nevertheless, these drawbacks were the genesis of the ideas used in CFLOBDDs:

- They showed that certain classes of cyclic structures, which contain an *infinite* number of paths, could still be interpreted as containing only a *finite* collection of realizable paths (defined by a suitable “matched-path” condition).
- Whereas one can sometimes obtain exponential compression when a collection of paths are represented as a DAG rather than as a tree, by passing to cyclic graphs augmented with a matched-path condition, one can sometimes obtain *doubly exponential compression* of a tree.

These properties suggested the basic structure of CFLOBDDs:

- Each level- i grouping in a CFLOBDD can be thought of as a loop-free procedure containing calls to level- $(i-1)$ groupings.
- Because a given grouping makes “calls” only on groupings at a strictly lower level, the “program” is non-recursive.

- Returns from procedure calls are somewhat non-standard in that “control” is not always returned to the “call-site.” Instead, returns in CFLOBDDs resemble *exit splitting* [14], where control can be returned to one of several return points in the caller.
- Such an interprocedural CFG is potentially cyclic, but contains only a finite number of realizable paths. In the *best case* (which corresponds to the *worst case* for path profiling), the number of realizable paths is doubly exponential in the size of the interprocedural CFG.

From here, it was only a short distance to CFLOBDDs—how to interpret such graphs as representations of Boolean functions; how to implement the standard BDD operations; how to maintain canonicalness; etc. The feature that threatened to sink the path-profiling scheme—doubly exponential *explosion*—became the linchpin of a doubly exponential *compressed* representation of Boolean functions. Metaphorically, a frog was turned into a prince.

Comparison with BDD variants. Over the years, many variants of BDDs have been proposed [67]. 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 [44] and FEVBDDs [72]. 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 Multi-Terminal BDDs [19, 21], Algebraic Decision Diagrams [7], Free Binary Decision Diagrams (FBDDs) [75, §6], Binary Moment Diagrams (BMDs) [18], Hybrid Decision Diagrams (HDDs) [20], Differential BDDs [6], and Indexed BDDs (IBDDs) [40]. Several of these BDD variants offer exponential compression over classical BDDs. However, because FBDDs, BMDs, HDDs, and IBDDs that encode, e.g., the identity function, need to examine each variable, the exponential-separation argument for CFLOBDDs from §8 carries over for all of these variants.

Cyclic BDD variants include Linear/Exponentially Inductive Functions (LIFs/EIFs) [34, 35] and Cyclic BDDs (CBDDs) [62]. Because they allow cycles, CFLOBDDs are closer to the structures in this category. The differences between CFLOBDDs and these representations can be characterized as follows:

- The aforementioned representations all make use of numeric/arithmetic annotations on the edges of the graphs used to represent functions over Boolean arguments, rather than the matched-path principle that is the basis of CFLOBDDs. Matched paths can be characterized in terms of a context-free language of matched parentheses, rather than in terms of numbers and arithmetic (see Eqn. (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 representations like Sentential Decision Diagrams (SDDs) [24] and Variable Shift SDDs [56]. 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 [41]) have not, so far, been used in matrix computations, and implementations of operations such as Kronecker product and matrix multiplication based on these structures are unknown, which meant that we could not use them in our quantum-simulation experiments. We did compare CFLOBDDs against SDDs for two of the micro-benchmarks, and found that CFLOBDDs were much faster (Tab. 2). However, the relationship between these representations and CFLOBDDs merits future study.

Relationship to PDSs, NWAs, etc. As noted in §1, CFLOBDDs can be seen as a generalization of BDDs in which a restricted form of procedure call is permitted. As such, CFLOBDDs are similar to various structures used in the model-checking community, namely, Hierarchical FSMs (HFSMs) [1], Push Down Systems (PDSs) [15, 27], and Nested-Word Automata (NWA) [3], which all have the same flavor of “graph plus procedure calls.”

- As discussed in Appendix §K, there is a formal sense in which a CFLOBDD is an NWA.
- HFSMs, PDSs, and NWAs have mainly been investigated for modeling infinite-state systems. What CFLOBDDs demonstrate is that there are advantages to considering finite restrictions of such structures. In particular, the advantage of introducing a procedure-call-like mechanism in a finite model is that one can obtain an exponential-factor advantage compared to the original finite models without procedure calls.

We believe that there is great potential for exploring other combinations of the idea of “BDDs plus procedure calls” that use ideas from HFSMs, PDSs, and NWAs in ways that are different than those found in CFLOBDDs.

CFLOBDDs can be considered to be a special case of Visibly Pushdown Automata (VPAs) [5] where A-connections and B-connections correspond to call transitions in a visibly pushdown language (VPL), return edges correspond to return transitions in the VPL, and edges at level-0 correspond to internal transitions of the VPL. There are other classes of VPAs, such as k -module single-entry VPAs (k -SEVPAs) [2] and Modular VPAs [43], that have minimal canonical representations. Such VPAs have modular components that are similar to the groupings in CFLOBDDs. However these classes of VPAs assume that the modular decomposition is fixed ahead of time. For instance, each k -SEVPA has exactly $k + 1$ modules. In contrast, CFLOBDDs use a decomposition that is based on a fixed number of levels, but the specific number of groupings for a function is a by-product of the structural invariants (§4.1 and Construction 1 in Appendix §C).

Prior Approaches to Quantum Simulation. Also related are prior methods for quantum simulation. Such simulation can be exact or approximate; our focus here is on exact simulation (modulo floating-point round-off). Decision diagrams used for such simulation include QMDDs [55, 83] and TDDs [36]. 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) [84, Tab. 5.1].³⁰ We also compared our approach to tensor networks, a widely used approach to quantum simulation that is not based on decision diagrams. As shown in Tab. 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 paper, variational quantum algorithms, which include a noise channel, can also be simulated using CFLOBDDs. Huang et al. [38] 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 nonrecursive 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 Melski’s Ph.D. thesis [52, §3.5.4]. Melski uses Yannakakis’s notion of L -reachability [80] (i.e., a path from node s to node t only counts as a valid s - t connection if the path’s labels form a word in L), and defines the notion of a “finite-path graph” with respect to some language L : there are only a finite number of L -paths from, e.g., program entry to program exit [52, §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 [53, §2.1], in which each return-edge is matched with the closest preceding unmatched call-edge, but there can be zero or more unmatched call-edges. (The unbalanced-left language is typically the language of interest for context-sensitive interprocedural dataflow analysis [63, 65].) Melski observes, “... the number of [unbalanced-left] paths through G_{fin}^* can be doubly exponential in the size of G_{fin}^* .”³¹

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 larger if written without non-recursive procedures.” In the 1970s, the literature on program schematology [60] 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 [48, 60]). Thus, it would have been natural for the schematology literature to contain a result of the form stated above. However, we were unable to find a paper with such a

³⁰Note that the number of qubits for Shor’s algorithm reported in [84, Tab. 5.1] is the number of qubits of the circuit, whereas in Tabs. 4 and 5, #Qubits is the number of bits of the number N being factored, where #Qubits-of-circuit = $2 * \#bits-of-N$.

³¹Unfortunately, the aforementioned work with the 20,000-line program that had $2^{400,000}$ paths (which is what prompted Melski and Reps to realize that they were facing double-exponential explosion) was carried out after their CC ’99 paper had been published [53]. The latter paper states, incorrectly, “In the worst case, the number of paths through a program is exponential in the number of branch statements $b \dots$ ” [53, §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 [57, 78] and version-space algebras [33, 42, 61].)

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 [23, 30].

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 $L(\text{matched})$ -paths is equal to the product of the numbers of $L(\text{matched})$ -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:

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

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 [49]. 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

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

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 $D_0, D_1 = \text{powerset}(D_0), \dots, D_n = \text{powerset}^n(D_0), \dots$, the size of the term D_n is $\Omega(n)$. In contrast, the size of the set that is represented by D_n is described by the following recurrence relation:

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

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

12 CONCLUSIONS AND FUTURE WORK

This paper described a new data structure—CFLOBDDs—for representing functions, matrices, relations, and other discrete structures. CFLOBDDs are a plug-compatible replacement for BDDs, and 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. The results presented in the paper include the following:

- We demonstrated the exponential separation between CFLOBDDs and BDDs both theoretically and experimentally.
- Because CFLOBDDs can be used as a replacement for BDDs, we gave algorithms for various operations that can be performed on representations of Boolean functions, such as Boolean and arithmetic operations, Kronecker product, matrix multiplication, etc.
- We also gave a set of structural invariants that ensure the canonicity of CFLOBDDs. When CFLOBDDs are implemented via hash-consing, this property ensures that the test of whether two CFLOBDDs represent equal functions can be performed merely by comparing the values of two pointers.
- We provided both an operational semantics and a denotational semantics of CFLOBDDs.
- We investigated the use of CFLOBDDs for representing the matrices and vectors involved in quantum simulation, and gave algorithms for the required operations.

In our experiments, we 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 is quite dramatic.

- For the micro-benchmarks, CFLOBDDs were able to represent versions of the benchmark functions with up to $1.5 \times 2^{8,388,609} \cong 1.2795 \times 10^{2,525,223}$ Boolean variables. In contrast, the largest BDD that we could construct successfully for any of the micro-benchmarks had about $328,000 = 3.28 \times 10^5$ Boolean variables.
- For the quantum-simulation benchmarks, the number of qubits that could be handled using CFLOBDDs was larger—compared to BDDs—by a factor of 128× for GHZ; 1,024× for BV; 8,192× for DJ; and 128× for Grover’s algorithm.

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.

Future Work. The work on CFLOBDDs opens up many avenues for future work. For instance, CFLOBDDs are based on the following decomposition pattern (or “pattern of calls”):

$$\begin{aligned} matched_k &\rightarrow matched_{k-1} \quad matched_{k-1} \\ matched_0 &\rightarrow \epsilon \end{aligned}$$

Other decomposition schemes are possible; for instance, in §8.3, it would have been useful to be able to define the ADD_n relation using the decomposition

$$\begin{aligned} matched_k &\rightarrow matched_{k-1} \quad matched_{k-1} \quad matched_{k-1} \\ matched_0 &\rightarrow \epsilon \end{aligned}$$

which would have avoided having to “waste” one-quarter of the variables, as in Figs. 16 and 17.

Another possibility would be to permit a level- i grouping to have connections to level- $i-j$ groupings, where $j \geq 1$). For instance, one could have a Fibonacci-like decomposition

$$\begin{aligned} matched_k &\rightarrow matched_{k-1} \quad matched_{k-2} \\ matched_1 &\rightarrow \epsilon \\ matched_0 &\rightarrow \epsilon \end{aligned}$$

As long as there is a fixed structural-decomposition pattern to how the levels connect, it should still possible to keep the same properties—i.e., the ability to perform operations implicitly, without having to instantiate the decision tree; canonicalness; etc.—as enjoyed by CFLOBDDs as defined in this paper (Defns. 3.1 and 4.1). We leave the exploration of such generalized CFLOBDDs for future work, along with such questions as

- How does one go about choosing a good structural-decomposition pattern for a given family of Boolean functions?
- Is it feasible for a CFLOBDD package to support the ability to change the structural-decomposition pattern dynamically (similar to the way that some BDD packages support the ability to change the variable ordering dynamically).

As mentioned in §11, there are other existing data structures [18, 20, 44, 72] that are exponentially more succinct than BDDs for some functions. However, the strategies used in the design of these data structures (for example, the use of a tree structure over variables in SDDs) are different than our strategy of exploiting the Matched-Path Principle. Bringing together these strategies and ours is an interesting avenue for future research.

Another natural direction is the study of *weighted* CFLOBDDs [70]. Also, while quantum simulation was the primary practical application studied in this paper, it is by no means the only

one. For example, CFLOBDDs may also have natural applications in settings like probabilistic programming, model checking, or circuit synthesis in which decision diagram representations have been successfully used in the past.

Finally, the idea of using programs with procedure calls as succinct data structures can be applied beyond our present setting of decision diagrams. In particular, automata-based representations of version spaces are a natural candidate for such additional compression. The exploration of these ideas in other settings is a rich and wholly open direction.

13 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 of Sistla et al. [69] 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 (Lem. L.2) does not imply that a global-reduction property holds (footnote 36), and for suggesting that we look for a polynomial-time bound in terms of the sizes of Reduce’s input and output proto-CFLOBDDs (§L).

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 Meghana Aparna Sistla. Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

REFERENCES

- [1] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *Trans. on Prog. Lang. and Syst.*, 27(4):786–818, 2005.
- [2] Rajeev Alur, Viraj Kumar, Partha Sarathy Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *International Colloquium on Automata, Languages, and Programming*, pages 1102–1114. Springer, 2005.
- [3] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.
- [4] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.
- [5] Rajeev Alur and Partha Sarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, 2004.
- [6] Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 218–233. Springer-Verlag, 1995.
- [7] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proc. of the Int. Conf. on Computer Aided Design*, pages 188–191, November 1993.
- [8] Thomas Ball and James R. Larus. Efficient path profiling. In *Proc. of MICRO-29*, December 1996.
- [9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001.
- [10] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE’01, Snowbird, Utah, USA, June 18-19, 2001*, pages 97–103. ACM, 2001.
- [11] Mari-Carmen Banuls, Matthew B Hastings, Frank Verstraete, and J Ignacio Cirac. Matrix product states for dynamical simulation of infinite chains. *Physical review letters*, 102(24):240603, 2009.
- [12] Stephane Beauregard. Circuit for Shor’s algorithm using $2n+3$ qubits. *arXiv preprint quant-ph/0205095*, 2002.
- [13] Michael Benedikt, Patrice Godefroid, and Thomas W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, pages 652–666, 2001.

- [14] Ras Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Prog. Lang. Design and Impl.*, pages 146–158, 1997.
- [15] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, 1997.
- [16] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th ACM/IEEE Design Automation Conf.*, pages 40–45, 1990.
- [17] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(6):677–691, August 1986.
- [18] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proc. of the 30th ACM/IEEE Design Automation Conf.*, pages 535–541, 1995.
- [19] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. Applications of multi-terminal binary decision diagrams. Technical Report CS-95-160, Carnegie Mellon Univ., School of Comp. Sci., April 1995.
- [20] Edmund M. Clarke, Masahiro Fujita, and Xudong Zhao. Hybrid decision diagrams: Overcoming the limitations of MTBDDs and BMDs. In *Proc. of the Int. Conf. on Computer Aided Design*, pages 159–163, November 1995.
- [21] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and Jerry Chih-Yuan Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. of the 30th ACM/IEEE Design Automation Conf.*, pages 54–60, 1993.
- [22] E.M. Clarke, M. Fujita, and X. Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In T. Sasao and M. Fujita, editors, *Representations of Discrete Functions*, pages 93–108. Kluwer Acad., Norwell, MA, 1996.
- [23] Robert L. Constable and David Gries. On classes of program schemata. *SIAM J. Comput.*, 1(1):66–118, 1972.
- [24] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.
- [25] Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. Abstract domains of affine relations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–73, 2014.
- [26] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19. ACM, 2006.
- [27] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
- [28] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Péllissier, and Paul Zimmermann. MPFR: A multiple-precision floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13–es, 2007.
- [29] Austin G. Fowler, Simon J. Devitt, and Lloyd C.L. Hollenberg. Implementation of Shor’s algorithm on a linear nearest neighbour qubit array. *arXiv preprint quant-ph/0402196*, 2004.
- [30] Stephen J. Garland and David C. Luckham. Program schemes, recursion schemes, and formal languages. *J. Comput. Syst. Sci.*, 7(2):119–160, 1973.
- [31] Eiichi Goto. Monocopy and associative algorithms in extended Lisp. Tech. Rep. TR 74-03, Univ. of Tokyo, Tokyo, Japan, 1974.
- [32] Johnnie Gray. quimb: A python library for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, 2018.
- [33] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017.
- [34] Aarti Gupta. *Inductive Boolean Function Manipulation: A Hardware Verification Methodology for Automatic Induction*. PhD thesis, Carnegie Mellon Univ., 1994. Tech. Rep. CMU-CS-94-208.
- [35] Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive Boolean functions. In *Proc. of the Int. Conf. on Computer Aided Design*, pages 192–199, November 1993.
- [36] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. A tensor network based decision diagram for representation of quantum circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2020.
- [37] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
- [38] Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. Logical abstractions for noisy variational quantum algorithm simulation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 456–472, 2021.
- [39] Stanley Leonard Hurst, D. Michael Miller, and Jon C. Muzio. *Spectral Techniques in Digital Logic*. Acad. Press, Inc., 1985.
- [40] Jawahar Jain, James R. Bitner, Magdy S. Abadir, Jacob A. Abraham, and Donald S. Fussell. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Trans. on Comp.*, C-46(11):1230–1245, November 1997.

- [41] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.
- [42] James Koppel. Version space algebras are acyclic tree automata. *CoRR*, abs/2107.12568, 2021.
- [43] Viraj Kumar, Partha Sarathy Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of Boolean programs. In *International Conference on Concurrency Theory*, pages 203–217. Springer, 2006.
- [44] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. of the 29th Conf. on Design Automation*, pages 608–613, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.
- [45] Ondrej Lhoták. *Program Analysis Using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.
- [46] Richard J. Lipton. BDD's and factoring, June 16, 2009. Gödel's Lost Letter and P=NP blog, <https://rjlipton.wordpress.com/2009/06/16/bdds-and-factoring/>.
- [47] Richard J. Lipton and Kenneth W. Regan. *Quantum Algorithms via Linear Algebra: A Primer*. MIT Press, 2014.
- [48] Nancy A. Lynch and Edward K. Blum. A difference in expressive power between flowcharts and recursion schemes. *Math. Syst. Theory*, 12:205–211, 1979.
- [49] Harry G. Mairson. A simple proof of a theorem of statman. *Theor. Comput. Sci.*, 103(2):387–394, 1992.
- [50] Wannes Meert and Arthur Choi. PySDD, v0.1. Zenodo, 10.5281/zenodo.1202374, March 2018.
- [51] David Melski. Personal communication, 1998.
- [52] David Melski. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation*. PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, February 2002. Tech. Rep. 1435.
- [53] David Melski and Thomas Reps. Interprocedural path profiling. In *Comp. Construct.*, pages 47–62, 1999.
- [54] Donald Michie. Memo functions: A language feature with ‘rote-learning’ properties. Technical Report MIP-R-29, Dept. of Machine Intelligence and Perception, Univ. of Edinburgh, Edinburgh, Scotland, November 1967.
- [55] D. Michael Miller and Mitchell A. Thornton. Qmdd: A decision diagram structure for reversible and quantum circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL '06)*, pages 30–30. IEEE, 2006.
- [56] Kengo Nakamura, Shuhei Denzumi, and Masaaki Nishino. Variable shift sdd: a more succinct sentential decision diagram. *arXiv preprint arXiv:2004.02502*, 2020.
- [57] C. Nandi, M. Willsey, A. Zhu, Y.R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock. Rewrite rule inference using equality saturation. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2021.
- [58] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.
- [59] Michael A Nielsen and Isaac L Chuang. Quantum computation and quantum information. *Phys. Today*, 54(2):60, 2001.
- [60] Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In Jack B. Dennis, editor, *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, Woods Hole, Massachusetts, USA, June 2–5, 1970, pages 119–127. ACM, 1970.
- [61] O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2015.
- [62] Frank Reffel. BDD-nodes can be more expressive. In *Proc. of the Asian Computing Science Conference*, December 1999.
- [63] T. Reps. Program analysis via graph reachability. In *Proc. of ILPS '97: Int. Logic Programming Symposium*, pages 5–19, Cambridge, MA, 1997. M.I.T.
- [64] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, pages 49–61, 1995.
- [65] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, pages 49–61, 1995.
- [66] Thomas W. Reps. Method for representing information in a highly compressed fashion, June 20, 2002. United States Patent Application 20020078431, filed Feb. 2, 2001, US Patent & Trademark Office. (Application abandoned.) <https://patentimages.storage.googleapis.com/ab/f4/17/2bbd2a0fad32f6/US20020078431A1.pdf>.
- [67] Tsutomu Sasao and Masahira Fujita, editors. *Representations of Discrete Functions*. Kluwer Acad., 1996.
- [68] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [69] Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. CFLOBDDs: Context-free-language ordered binary decision diagrams. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–73, 2024. To appear; DOI: 10.1145/3651157.
- [70] Meghana Sistla, Swarat Chaudhuri, and Thomas W. Reps. Weighted context-free-language ordered binary decision diagrams. *CoRR*, abs/2305.13610, 2023.
- [71] Fabio Somenzi. CUDD: CU decision diagram package—release 2.4.0. *University of Colorado at Boulder*, 2012.
- [72] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2):243–270, 1997.

- [73] Frank Verstraete, Juan J Garcia-Ripoll, and Juan Ignacio Cirac. Matrix product density operators: Simulation of finite-temperature and dissipative systems. *Physical review letters*, 93(20):207204, 2004.
- [74] Guifré Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical review letters*, 91(14):147902, 2003.
- [75] Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM Monographs on Disc. Math. and Appl. Society for Industrial and Applied Mathematics, 2000.
- [76] J. Whaley, D. Avots, M. Carbin, and M.S. Lam. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems*, 2005.
- [77] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Prog. Lang. Design and Impl.*, 2004.
- [78] M. Willsey. Fast and extensible equality saturation with egg, 2021.
- [79] Kieran Woolfe. *Matrix product operator simulations of quantum algorithms*. PhD thesis, University of Melbourne, School of Physics, 2015.
- [80] M. Yannakakis. Graph-theoretic methods in database theory. In *Symp. on Princ. of Database Syst.*, pages 230–242, 1990.
- [81] Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 542–558, 2021.
- [82] Xusheng Zhi and Thomas Reps. Polynomial bounds of CFLOBDDs against BDDs. In preparation, May 2024.
- [83] Alwin Zulehner and Robert Wille. Advanced simulation of quantum computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 38(5):848–859, 2019.
- [84] Alwin Zulehner and Robert Wille. *Introducing Design Automation for Quantum Computing*. Springer, 2020.

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 ints—and, in particular, if $f(3)$ returns a pair consisting of the values 4 and 5—then int variables a and b would be assigned 4 and 5 by the following initialized declaration:

```
int×int [a,b] = f(3)
```

- The indices of array elements start at 1.
- Arrays are allocated with an initial length (which is allowed to be 0); however, arrays are assumed to lengthen automatically to accommodate assignments at index positions beyond the current length.
- We assume that a call on the constructor `InternalGrouping(k)` returns an `InternalGrouping` in which the members have been initialized as follows:

```
level = k
AConnection = NULL
AReturnTuple = NULL
numberOfBConnections = 0
BConnections = new array[0] of Grouping
BReturnTuples = new array[0] of ReturnTuple
numberOfExits = 0
```

Similarly, we assume that a call on the constructor `CFLOBDD(g, vt)` returns a `CFLOBDD` in which the members have been initialized as follows:

```
grouping = g
valueTuple = vt
```

The class definitions of Fig. 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 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 the proposition holds for every level- k proto-CFLOBDD.

Let C be an arbitrary level- $k+1$ proto-CFLOBDD, with s and ex_C as defined above. Without loss of generality, we will refer to the exit vertices by ordinal position; i.e., we will consider ex_C to be the sequence $[1, 2, \dots, |ex_C|]$. 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 concatenation of assignments (e.g., $\alpha_j = \alpha_j^1 + \alpha_j^2$).

There are two cases to consider.

Case 1: $\alpha_i^1 = \alpha_j^1$ and $\alpha_i^2 \prec \alpha_j^2$.

Because $\alpha_i^1 = \alpha_j^1$, the first halves of the matched path followed during the interpretations of assignments α_i and α_j through C_A are identical, and bring us to some middle vertex, say m , of C ; both paths then proceed through C_{B_m} . Let e_i and e_j be the two exit vertices of C_{B_m} reached by following matched paths during the interpretations of α_i^2 and α_j^2 , respectively. There are now two cases to consider:

Case 1.A: Suppose that $e_i < e_j$ in C_{B_m} (see Fig. 27a). In this case, the return edges $e_i \rightarrow i$ and $e_j \rightarrow j$ “cross”. By Structural Invariant 2b, this can only happen if

- There is a matched path corresponding to some assignment β^1 through C_A that leads to a middle vertex h , where $h < m$.
- There is a matched path from h corresponding to some assignment β^2 through C_{B_h} (where C_{B_h} could be C_{B_m}).
- There is a return edge from the exit vertex reached by β^2 in C_{B_h} to exit vertex j of C .

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

Consequently, $\beta^1 + \beta^2 \prec \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j .

Case 1.B: Suppose that $e_j < e_i$ in C_{B_m} (see Fig. 27b). Because $\alpha_i^2 \prec \alpha_j^2$, the induction hypothesis applied to C_{B_m} implies that there must exist an assignment $\gamma \prec \alpha_i^2 \prec \alpha_j^2$ that leads to e_j . In this case, we have that $\alpha_j^1 + \gamma \prec \alpha_j^1 + \alpha_j^2$, which again contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j .

Case 2: $\alpha_i^1 \prec \alpha_j^1$.

Because $\alpha_i^1 \prec \alpha_j^1$, the first halves of the matched paths followed during the interpretations of assignments α_i and α_j through C_A bring us to two different middle vertices of C , say m and n , respectively. The two paths then proceed through C_{B_m} and C_{B_n} (where it could be the case that $C_{B_m} = C_{B_n}$), and return to i and j , respectively, where $j < i$. Again, there are two cases to consider:

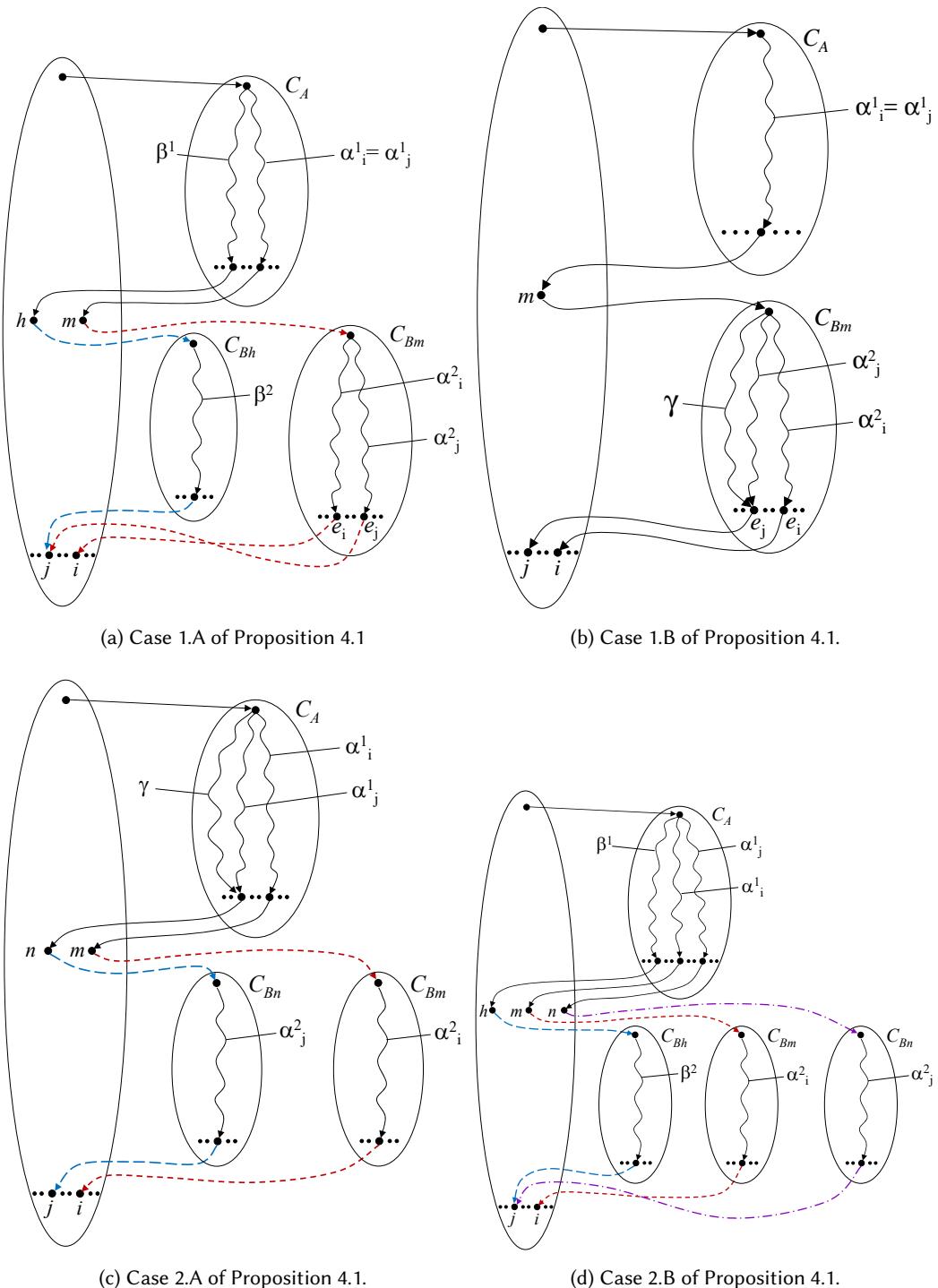


Fig. 27

Case 2.A: Suppose that $n < m$ (see Fig. 27c.) The argument is similar to Case 1.B above: By Structural Invariant 1, $n < m$ means that the exit vertex reached by α_j^1 in C_A comes before the exit vertex reached by α_i^1 in C_A . By the induction hypothesis applied to C_A , there must exist an assignment $\gamma < \alpha_i^1 < \alpha_j^1$ that leads to the exit vertex reached by α_j^1 in C_A . In this case, we have that $\gamma + \alpha_j^2 < \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j .

Case 2.B: Suppose that $m < n$ (see Fig. 27d.) The argument is similar to Case 1.A above: By Structural Invariant 2, we can only have $m < n$ and $j < i$ if

- There is a matched path corresponding to some assignment β^1 through C_A that leads to a middle vertex h , where $h < m$.
- There is a matched path from h corresponding to some assignment β^2 through C_{B_h} (where C_{B_h} could be C_{B_m} or C_{B_n}).
- There is a return edge from the exit vertex reached by β^2 in C_{B_h} to exit vertex j of C .

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

Consequently, $\beta^1 + \beta^2 < \alpha_j^1 + \alpha_j^2$, which contradicts the assumption that $\alpha_j = \alpha_j^1 + \alpha_j^2$ is the least assignment in lexicographic order that leads to j .

In each of the cases above, we are able to derive a contradiction to the assumption that α_j is the least assignment in lexicographic order that leads to j . Thus, the supposition that the proposition does not hold for C cannot be true. \square

C PROOF OF THE CANONICALNESS OF CFLOBDDS

To show that CFLOBDDs are a canonical representation of functions over Boolean arguments, we must establish that three properties hold:

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

As described earlier, following a matched path (of length $O(2^k)$) from the level- k entry vertex of a level- k CFLOBDD to a final value provides an interpretation of a Boolean assignment on 2^k variables. Thus, the CFLOBDD represents a decision tree with 2^{2^k} leaves (and Obligation 1 is satisfied).

To show that Obligation 2 holds, we describe a recursive procedure for constructing a level- k CFLOBDD from an arbitrary decision tree with 2^{2^k} leaves (i.e., of height 2^k). In essence, the construction shows how such a decision tree can be folded together to form a multi-terminal CFLOBDD.

The construction makes use of a set of auxiliary tables, one for each level, in which a unique representative for each class of equal proto-CFLOBDDs that arises is tabulated. We assume that the level-0 table is already seeded with a representative fork grouping and a representative don't-care grouping.

CONSTRUCTION 1. [Decision Tree to Multi-Terminal CFLOBDD]

- (1) *The leaves of the decision tree are partitioned into some number of equivalence classes e according to the values that label the leaves. The equivalence classes are numbered 1 to e according to the relative position of the first occurrence of a value in a left-to-right sweep over the leaves of the decision tree.*

For Boolean-valued CFLOBDDs, when the procedure is applied at topmost level, there are at most two equivalence classes of leaves, for the values F and T. However, in general, when the procedure is applied recursively, more than two equivalence classes can arise.

For the general case of multi-terminal CFLOBDDs, the number of equivalence classes corresponds to the number of different values that label leaves of the decision tree.

- (2) (**Base cases**) If $k = 0$ and $e = 1$, construct a CFLOBDD consisting of the representative don't-care grouping, with a value tuple that binds the exit vertex to the value that labels both leaves of the decision tree.

If $k = 0$ and $e = 2$, construct a CFLOBDD consisting of the representative fork grouping, with a value tuple that binds the two exit vertices to the first and second values, respectively, that label the leaves of the decision tree.

If either condition applies, return the CFLOBDD so constructed as the result of this invocation; otherwise, continue on to the next step.

- (3) Construct—via recursive applications of the procedure— $2^{2^{k-1}}$ level- $k-1$ multi-terminal CFLOBDDs for the $2^{2^{k-1}}$ decision trees of height 2^{k-1} in the lower half of the decision tree.

These are then partitioned into some number e' of equivalence classes of equal multi-terminal CFLOBDDs; a representative of each class is retained, and the others discarded. Each of the $2^{2^{k-1}}$ “leaves” of the upper half of the decision tree is labeled with the appropriate equivalence-class representative for the subtree of the lower half that begins there. These representatives serve as the “values” on the leaves of the upper half of the decision tree when the construction process is applied recursively to the upper half in step 4.

The equivalence-class representatives are also numbered 1 to e' according to the relative position of their first occurrence in a left-to-right sweep over the leaves of the upper half of the decision tree.

- (4) Construct—via a recursive application of the procedure—a level- $k-1$ multi-terminal CFLOBDD for the upper half of the decision tree.
- (5) Construct a level- k multi-terminal proto-CFLOBDD from the level- $k-1$ multi-terminal CFLOBDDs created in steps 3 and 4. The level- k grouping is constructed as follows:
- (a) The A-connection points to the proto-CFLOBDD of the object constructed in step 4.
 - (b) The middle vertices correspond to the equivalence classes formed in step 3, in the order $1 \dots e'$.
 - (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 $1 \dots e'$.
 - (e) The exit vertices correspond to the initial equivalence classes described in step 1, in the order $1 \dots e$.
 - (f) The B-connection return tuples connect the exit vertices of the highest-level groupings of the equivalence-class representatives retained from step 3 to the exit vertices created in step 5e. In each of the equivalence-class representatives retained from step 3, the value tuple associates each exit vertex x with some value v , where $1 \leq v \leq e$; x is now connected to the exit vertex created in step 5e that is associated with the same value v .
 - (g) Consult a table of all previously constructed level- k groupings to determine whether the grouping constructed by steps 5a–5f duplicate a previously constructed grouping. If so, discard the present grouping and switch to the previously constructed one; if not, enter the present grouping into the table.

- (6) *Return a multi-terminal CFLOBDD created from the proto-CFLOBDD constructed in step 5 by attaching a value tuple that connects (in order) the exit vertices of the proto-CFLOBDD to the e values from step 1.*

□

Fig. 28a shows the decision tree for the function $\lambda x_0 x_1 x_2 x_3. (x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$. Fig. 28b 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,³² the leftmost proto-CFLOBDD maps its exit vertex to F , whereas the exit vertex is mapped to T in the second and third proto-CFLOBDDs. Thus, in this case, the recursive call for the upper half of the decision tree (step 4) involves three equivalence classes of values.

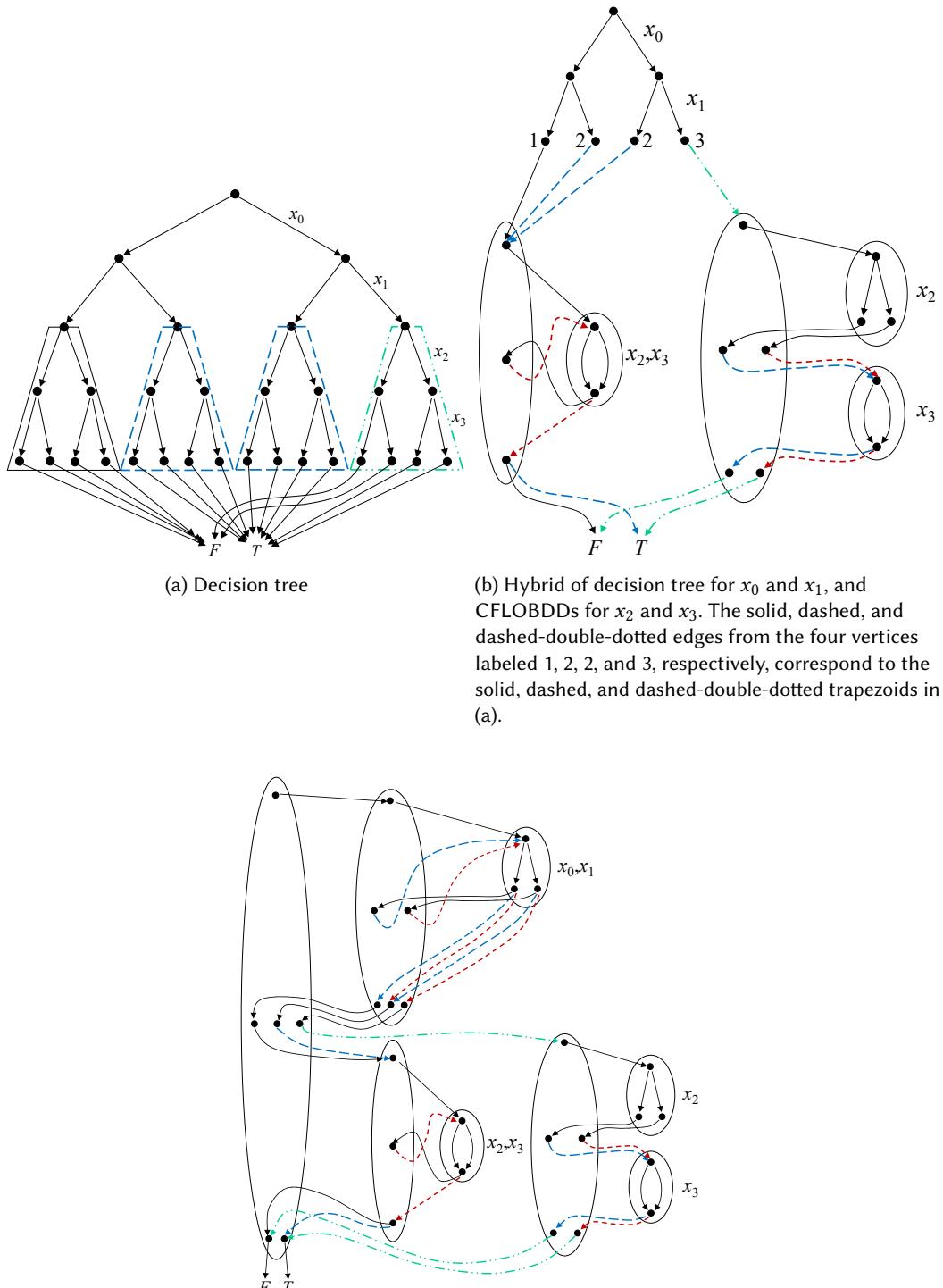
It is not hard to see that the structures created by Construction 1 obey the structural invariants that are required of CFLOBDDs:

- Structural Invariant 1 holds because the A -connection return tuple created in step 5c of Construction 1 is the identity map.
- Structural Invariant 2 holds because in steps 1 and 3 of Construction 1, the equivalence classes are numbered in increasing order according to the relative position of a value's first occurrence in a left-to-right sweep. In particular, this order is preserved in the exit vertices of each grouping constructed during an invocation of Construction 1 (cf. step 5f), which ensures that the “compact extension” property of Structural Invariant 2b holds at each level of recursion in Construction 1.
- 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, \dots, 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*,

³²The equality of the proto-CFLOBDDs is detected in step 5g.



(c) CFLOBDD (repeated from Fig. 5). For clarity, some of the level-0 groupings have been duplicated.

Fig. 28. Representations of the Boolean function $\lambda x_0 x_1 x_2 x_3. (x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$.

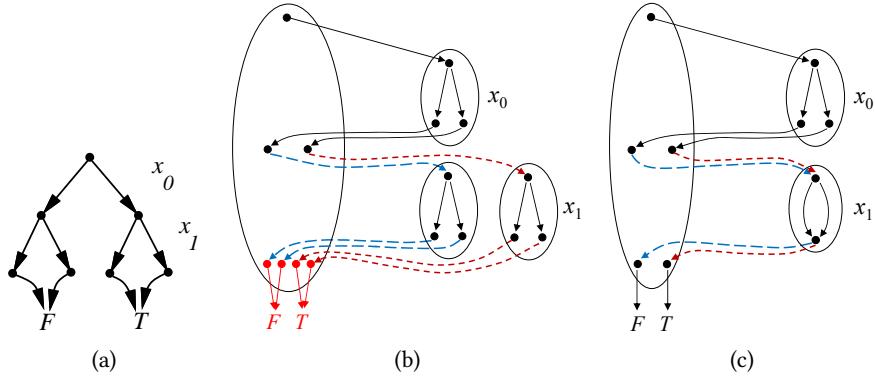


Fig. 29. (a) Decision tree for $\lambda x_0x_1.x_0$; (b) fully expanded form of the CFLOBDD; (c) CFLOBDD.

Trace for entire tree				
Tree	The lower-half trees	Hybrid	Upper-half tree	CFLOBDD

Fig. 30. The *Fold* trace generated by the application of Construction 1 to the decision tree shown in Fig. 29a to create the CFLOBDD shown in Fig. 29c.

that starts with a multi-terminal CFLOBDD and works in the opposite direction to Construction 1 to construct a decision tree; that is, *Unfold* (recursively) unfolds the *A*-connection, and then (recursively) unfolds each of the *B*-connections. For instance, for the example shown in Fig. 28, *Unfold* would proceed from Fig. 28c, to Fig. 28b, and then to the decision tree for the function $\lambda x_0x_1x_2x_3.(x_0 \oplus x_1) \vee (x_0 \wedge x_1 \wedge x_2)$ shown in Fig. 28a.

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, \dots, 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 .)

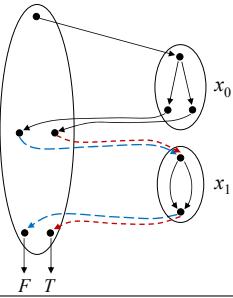
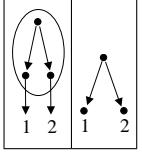
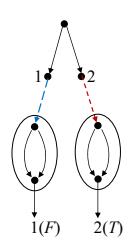
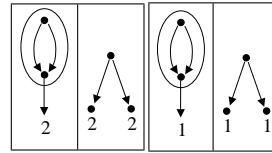
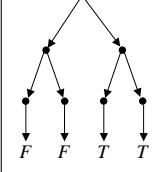
Trace for entire CFLOBDD				
CFLOBDD	A-connection	Hybrid	Traces for the <i>B</i> -connections	Tree
				

Fig. 31. The *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Fig. 29c to create the decision tree shown in Fig. 29a.

Obligation 3 is satisfied if we can show that, for every CFLOBDD C , Construction 1 applied to the decision tree produced by $\text{Unfold}(C)$ 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 Fig. 28b) is appended to the trace.
 - During step 4, the trace is extended according to the actions carried out by the folding process as it is applied recursively to the upper half of the decision tree.
 - At the end of step 6, the CFLOBDD being returned is appended to the trace.
 For instance, Fig. 30 shows the *Fold* trace generated by the application of Construction 1 to the decision tree shown in Fig. 29a to create the CFLOBDD shown in Fig. 29c.
- An *Unfold trace* records the steps of $\text{Unfold}(C)$:
 - CFLOBDD C is appended to the trace.
 - If C is a level-0 CFLOBDD, then a binary tree of height 1—with the leaves labeled according to C 's value tuple—is appended to the trace (and the *Unfold* algorithm returns).
 - The trace is extended according to the actions carried out by *Unfold* as it is applied recursively to the *A*-connection of C .
 - A hybrid decision-tree/CFLOBDD object (à la Fig. 28b) is appended to the trace.
 - The trace is extended according to the actions carried out by *Unfold* as it is applied recursively to instances of *B*-connections of C . (For purposes of settling Obligation 3, we will assume that *Unfold* processes a separate instance of a *B*-connection for each leaf of the hybrid object's upper-half decision tree, and that the *B*-connections are processed in *right-to-left* order of the upper-half decision tree's leaves.)

– Finally, the decision tree returned by *Unfold* is appended to the trace.

For instance, Fig. 31 shows the *Unfold* trace generated by the application of *Unfold* to the CFLOBDD shown in Fig. 29c to create the decision tree shown in Fig. 29a.

Note how the *Unfold* trace shown in Fig. 31 is the reversal of the *Fold* trace shown in Fig. 30. We now argue that this property holds generally. (Technically, the argument given below in Proposition C.1 shows that each element of an *Unfold* trace is *isomorphic* to the corresponding object in the *Fold* trace, which suffices to imply that that Obligation 3 is satisfied, in the sense that a decision tree is represented by exactly one isomorphism class of CFLOBDDs.)

PROPOSITION C.1. *Suppose that C is a multi-terminal CFLOBDD, and that $\text{Unfold}(C)$ results in Unfold trace UT and decision tree T_0 . Let C' be the multi-terminal CFLOBDD produced by applying Construction 1 to T_0 , and FT be the *Fold* trace produced during this process. Then*

- (i) FT is the reversal of UT .
- (ii) C and C' are isomorphic.

Proof: Because C' appears at the end of FT , and C appears at the beginning of UT , clause (i) implies (ii). We show clause (i) by the following inductive argument:

Base case: The proposition is trivially true of level-0 CFLOBDDs. Given any pair of values v_1 and v_2 (such as F and T), there are exactly four possible level-0 CFLOBDDs: two constructed using a don't-care grouping—one in which the exit vertex is mapped to v_1 , and one in which it is mapped to v_2 —and two constructed using a fork grouping—one in which the two exit vertices are mapped to v_1 and v_2 , respectively, and one in which they are mapped to v_2 and v_1 , respectively. These unfold to the four decision trees that have $2^{2^0} = 2$ leaves and leaf-labels drawn from $\{v_1, v_2\}$, and the application of Construction 1 to these decision trees yields the same level-0 CFLOBDD that we started with. (See step 2 of Construction 1.) Consequently, the *Fold* trace FT and the *Unfold* trace UT are reversals of each other.

Induction step: The induction hypothesis is that the proposition holds for every level- k multi-terminal CFLOBDD. We need to argue that the proposition extends to level- $k+1$ multi-terminal CFLOBDDs.

First, note that the induction hypothesis implies that each decision tree with 2^{2^k} leaves is represented by exactly one level- k CFLOBDD isomorphism class. We will refer to this as the *corollary to the induction hypothesis*.

Unfold trace UT can be divided into five segments:

- (u1) C itself
- (u2) the *Unfold* trace for C 's A -connection
- (u3) a hybrid decision-tree/CFLOBDD object (call this object D)
- (u4) the *Unfold* trace for C 's B -connections
- (u5) T_0 .

Fold trace FT can also be divided into five segments:

- (f1) T_0
- (f2) the *Fold* trace for T_0 's lower-half trees
- (f3) a hybrid decision-tree/CFLOBDD object (call this object D')
- (f4) the *Fold* trace for T_0 's upper-half
- (f5) C' .

Because both (f1) and (u5) are T_0 , (u5) is obviously equal to (f1). Our goal, therefore, is to show that

- (u2) is the reversal of (f4);
- (u3) is equal to (f3);
- (u4) is the reversal of (f2); and
- (u1) is equal to (f5).

(u3) is equal to (f3) Consider the hybrid decision-tree/CFLOBDD object D obtained after *Unfold* has finished unfolding C 's A -connection.³³ 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 Fig. 28b).

As a consequence of Prop. 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.³⁴ However, each middle vertex has an associated B -connection, and by Structural Invariants 2, 4, and 6, the middle vertices can be thought of as representatives for a set of pairwise non-equal CFLOBDDs (that themselves represent lower-half decision trees).

Fold trace FT also has a hybrid decision-tree/CFLOBDD object, namely D' . The crucial point is that the action of partitioning T_0 's lower-half CFLOBDDs that is carried out in step 3 of Construction 1 also results in a labeling of each leaf of the upper-half's decision tree with a representative of an equivalence class of CFLOBDDs that represent the lower half of the decision tree starting at that point.

By the corollary to the induction hypothesis, the 2^{2^k} bottom-half trees of T_0 are represented uniquely (up to isomorphism) by the respective CFLOBDDs in D' . Similarly, by the corollary to the induction hypothesis, the 2^{2^k} CFLOBDDs used as labels in D represent uniquely (up to isomorphism) the respective bottom-half trees of T_0 . Thus, the labelings on D and D' must be isomorphic.

(u2) is the reversal of (f4); (u4) is the reversal of (f2) Given the observation that D and D' are isomorphic, these properties follow in a straightforward fashion from the inductive hypothesis (applied to the A -connection and the B -connections of C).

(u1) is equal to (f5) Because (u2) is the reversal of (f4) and (u4) is the reversal of (f2), we know that the level- k proto-CFLOBDDs out of which the level- $k+1$ grouping of C' is constructed are isomorphic to the respective level- k proto-CFLOBDDs that make up the A -connection and B -connections of C .

We already argued that steps 5 and 6 of Construction 1 lead to CFLOBDDs that obey the six structural invariants required of CFLOBDDs. Moreover, there is only one way for Construction 1 to construct the level- $k+1$ grouping of C' so that Structural Invariants 2, 3, and 4 are satisfied. Therefore, C is isomorphic to C' .

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

In summary, we have now shown that Obligations 1, 2, and 3 are all satisfied. These properties imply that, for a given ordering of Boolean variables, if two level- k CFLOBDDs C_1 and C_2 represent the same decision tree with 2^{2^k} leaves, then C_1 and C_2 are isomorphic—i.e., CFLOBDDs are a canonical representation of functions over Boolean arguments:

³³The 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.

³⁴This step is where the argument would break down if we attempt to apply the same argument to Fig. 8a: In that case, the labels on the leaves of D , in left-to-right order, would be 2 and 1—whereas the sequence of middle vertices in Fig. 8a is [1,2].

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.

D PAIR PRODUCT CANONICITY PROOF

We prove that the grouping g constructed during a call on `PairProduct` meets Structural Invariant 4—and hence it is permissible to call `RepresentativeGrouping(g)` in line [38] of Alg. 12.

In particular, suppose that (i) B_1 and B'_1 are B -connections of a grouping g_1 (with associated return tuples rt_1 and rt'_1 , respectively), (ii) B_2 and B'_2 are B -connections of a grouping g_2 (with associated return tuples rt_2 and rt'_2 , respectively), and (iii) at least one of the following two conditions hold:

- (1) $\langle B_1, rt_1 \rangle \neq \langle B'_1, rt'_1 \rangle$
- (2) $\langle B_2, rt_2 \rangle \neq \langle B'_2, rt'_2 \rangle$

In addition, suppose that the recursive calls on `PairProduct` produce

$$[D, pt] = \text{PairProduct}(B_1, B_2) \quad \text{and} \quad [D', pt'] = \text{PairProduct}(B'_1, B'_2),$$

Let rt and rt' be the return tuples that the outer calls on `PairProduct` in line [22] of Alg. 12 create for D and D' : pt , rt_1 , and rt_2 are used to create rt ; pt' , rt'_1 , and rt'_2 are used to create rt' .

The question that we need to answer is whether it is ever possible for both $D = D'$ and $rt = rt'$ to hold. This question is of concern because the hypothesized condition would violate Structural Invariant 4: if the condition were to hold, then the first entry of the pair returned by `PairProduct` would not be a well-formed proto-CFLOBDD. The following proposition shows that, in fact, this situation cannot ever occur:

PROPOSITION D.1. *The first entry of the pair returned by `PairProduct` is always a well-formed proto-CFLOBDD.*

Proof: We argue by induction:

Base case: When g_1 and g_2 are level-0 groupings, there are four cases to consider. In each case, it is immediate from lines [2]–[5] of Alg. 11 that the first entry of the pair returned by `PairProduct` is a well-formed proto-CFLOBDD.

Induction step: The induction hypothesis is that the first entry of the pair returned by `PairProduct` is a well-formed proto-CFLOBDD whenever the arguments to `PairProduct` are level- k proto-CFLOBDDs.

Let g_1 and g_2 be two arbitrary well-formed level- $k+1$ proto-CFLOBDDs. We argue by contradiction: suppose, for the sake of argument, that D , D' , rt , and rt' are as defined above, and that both $D = D'$ and $rt = rt'$ hold.

- By the inductive hypothesis, we know that D and D' are each well-formed level- k proto-CFLOBDDs. In particular, we can think of D and rt as corresponding to a decision tree T_0 , labeled with the exit vertices of g to which the decision tree's leaves are mapped. However, because of the search that is carried out in lines [24]–[36] of `PairProduct` (Alg. 12), each exit vertex of g corresponds to a unique pair, $\langle c_1, c_2 \rangle$, where c_1 and c_2 are exit vertices of g_1 and g_2 , respectively. Thus, a leaf in T_0 can be thought of as being labeled with a pair $\langle c_1, c_2 \rangle$.

Furthermore, because $D = D'$ and $rt = rt'$, D' and rt' also correspond to decision tree T_0 .

- When T_0 is considered to be the decision tree associated with D and rt , we can read off (a) the decision tree that corresponds to B_1 with exit vertices of g_1 labeling the leaves (call this tree T_1), and (b) the decision tree that corresponds to B_2 with exit vertices of g_2 labeling the leaves

(T_2) . Similarly, when T_0 is considered to be the decision tree associated with D' and rt' , we can read off (c) the decision tree that corresponds to B'_1 with exit vertices of g_1 labeling the leaves (T'_1), and (d) the decision tree that corresponds to B'_2 with exit vertices of g_2 labeling the leaves (T'_2). (We use the first entry of each $\langle c_1, c_2 \rangle$ pair for B_1 and B'_1 , and the second entry of each $\langle c_1, c_2 \rangle$ pair for B_2 and B'_2 .) This process gives us four trees, T_1 , T'_1 , T_2 , and T'_2 , where—from the supposition that $D = D'$ and $rt = rt'$ —we must have $T_1 = T'_1$ and $T_2 = T'_2$.

- By assumption, g_1 and g_2 are well-formed proto-CFLOBDDs; thus, by Structural Invariant 2, all return tuples for the B -connections of g_1 and g_2 must represent 1-to-1 maps. Moreover, B_1 , B_2 , B'_1 , and B'_2 are also well-formed proto-CFLOBDDs, which means that, in g_1 , B_1 together with rt_1 must be the unique representative of occurrences of T_1 in g_1 's decision tree, while B'_1 together with rt'_1 must be the unique representative of occurrences of T'_1 . Similarly, in g_2 , B_2 together with rt_2 must be the unique representative of occurrences of T_2 in g_2 's decision tree, while B'_2 together with rt'_2 must be the unique representative of occurrences of T'_2 .

Therefore, in g_1 , we have

– $B_1 = B'_1$ and $rt_1 = rt'_1$,

while in g_2 , we have

– $B_2 = B'_2$ and $rt_2 = rt'_2$.

However, these conditions are a violation of Structural Invariant 4, which, in turn, contradicts the assumption that g_1 and g_2 are well-formed level- $k+1$ proto-CFLOBDDs. Consequently, the assumption that $D = D'$ and $rt = rt'$ cannot be true.

□

E ADDITIONAL OPERATIONS ON CFLOBDDS

E.1 Ternary Operations on CFLOBDDs

This section discusses how ternary operations (i.e., three-argument operations) on CFLOBDDs are performed. Algs. 26 and 27 present the two algorithms needed to implement ternary operations on multi-terminal CFLOBDDs. As in §7.3, we assume that the CFLOBDD or Grouping arguments of the operations described below are objects whose highest-level groupings are all at the same level.

- The operation `TernaryApplyAndReduce` given in Alg. 26 is very much like the operation `BinaryApplyAndReduce` of Alg. 10, except that it starts with a call on `TripleProduct` instead of `PairProduct` (line [3]).
- The operation `TripleProduct`, which is given in Alg. 27, is very much like the operation `PairProduct` of Alg. 11, except that `TripleProduct` has a third Grouping argument, and performs a three-way—rather than two-way—cross product of the three Grouping arguments: g_1 , g_2 , and g_3 . `TripleProduct` returns the proto-CFLOBDD g formed in this way, as well as a descriptor of the exit vertices of g in terms of triples of exit vertices of the highest-level groupings of g_1 , g_2 , and g_3 .

(By an argument similar to the one given for `PairProduct`, it is possible to show that the grouping g constructed during a call on `TripleProduct` is always a well-formed proto-CFLOBDD—and hence it is permissible to call `RepresentativeGrouping(g)` in line [55] of Alg. 27.)

- `TernaryApplyAndReduce` then uses the triples describing the exit vertices to determine the tuple of leaf values that should be associated with the exit vertices (i.e., a tentative value tuple) (line [4]).
- Finally, `TernaryApplyAndReduce` proceeds in the same manner as `BinaryApplyAndReduce`:
 - Two tuples that describe the collapsing of duplicate leaf values—assuming folding to the left—are created via a call to `CollapseClassesLeftmost` (line [5]).

Algorithm 26: TernaryApplyAndReduce

```

Input: CFLOBDDs n1, n2, n3, Op op
Output: CFLOBDD n = op(n1, n2, n3)
1 begin
2   assert(n1.grouping.level == n2.grouping.level && n2.grouping.level ==
      n3.grouping.level);
      // Perform triple cross product
3   Grouping×TripleTuple [g,tt] = TripleProduct(n1.grouping, n2.grouping, n3.grouping);
      // Create tuple of "leaf" values
4   ValueTuple deducedValueTuple = [op(n1.valueTuple[i1], n2.ValueTuple[i2],
      n3.ValueTuple[i3]) : [i1,i2,i3] ∈ tt];
      // Collapse duplicate leaf values, folding to the left
5   Tuple×Tuple [inducedValueTuple,inducedReturnTuple] =
      CollapseClassesLeftmost(deducedValueTuple);
      // Perform corresponding reduction on g, folding g's exit vertices
      w.r.t. inducedReturnTuple
6   Grouping g' = Reduce(g, inducedReturnTuple);
7   return RepresentativeCFLOBDD(g', inducedValueTuple);
8 end

```

- The corresponding reduction is performed on Grouping g, by calling Reduce to fold g’s exit vertices with respect to variable inducedReturnTuple (one of the tuples returned by the call on CollapseClassesLeftmost) (line [6]).

Lastly, in the case of Boolean-valued CFLOBDDs, there are 256 possible ternary operations, corresponding to the 256 possible three-argument truth tables ($2 \times 2 \times 2$ matrices with Boolean entries). All 256 possible ternary operations are special cases of TernaryApplyAndReduce; these can be performed by passing TernaryApplyAndReduce an appropriate value for argument op (i.e., some $2 \times 2 \times 2$ Boolean matrix).

One of the 256 ternary operations is the operation called ITE [16] (for “If-Then-Else”), which is defined as follows:

$$\text{ITE}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c).$$

Appendix §F shows how the ternary ITE operation can be used to implement all 16 of the binary operations on Boolean-valued CFLOBDDs [16].

E.2 Restrict

We now discuss the restriction operation. Given a value v to which variable x_i is to be bound (e.g., by giving either the assignment $[x_i \mapsto T]$ or the assignment $[x_i \mapsto F]$), the Restrict operation applies the assignment to the CFLOBDD that represents a function f , and returns the CFLOBDD that represents $f|_{x_i=v}$. Algs. 29 and 30 gives pseudo-code for the algorithm. At each level, the algorithm checks if index i belongs to the *A-Connections* or *B-Connections* at every level, and calls Restrict recursively, as appropriate, on lower levels with an adjusted index value i . The rest of the groupings are kept as is, except that some groupings are eliminated, and the positions of the remaining ones shifts (Alg. 30, lines [3] and [15]).

Algorithm 27: TripleProduct

Input: Groupings g_1, g_2, g_3

Output: Grouping g : product of g_1, g_2, g_3 ; TripleTuple $ttAns$: tuple of triples of exit vertices

```

1 begin
2   if  $g_1, g_2, g_3$  are all no-distinction proto-CFLOBDDs then
3     | return  $[g_1, [[1,1,1]]]$ ;
4   end
5   if  $g_1$  and  $g_2$  no-distinction proto-CFLOBDDs then
6     | return  $[g_3, [[1,1,k]: k \in [1..g_3.numberOfExits]]]$ ;
7   end
8   if  $g_1$  and  $g_3$  no-distinction proto-CFLOBDDs then
9     | return  $[g_2, [[1,k,1]: k \in [1..g_2.numberOfExits]]]$ ;
10  end
11  if  $g_2$  and  $g_3$  no-distinction proto-CFLOBDDs then
12    | return  $[g_1, [[k,1,1]: k \in [1..g_1.numberOfExits]]]$ ;
13  end
14  if  $g_1$  is a no-distinction proto-CFLOBDD then
15    | Grouping $\times$ PairTuple  $[g, pt] = \text{PairProduct}(g_2, g_3)$ ;
16    | return  $[g, [[1,j,k]: [j,k] \in pt]]$ ;
17  end
18  if  $g_2$  is a no-distinction proto-CFLOBDD then
19    | Grouping $\times$ PairTuple  $[g, pt] = \text{PairProduct}(g_1, g_3)$ ;
20    | return  $[g, [[j,1,k]: [j,k] \in pt]]$ ;
21  end
22  if  $g_3$  is a no-distinction proto-CFLOBDD then
23    | Grouping $\times$ PairTuple  $[g, pt] = \text{PairProduct}(g_1, g_2)$ ;
24    | return  $[g, [[j,k,1]: [j,k] \in pt]]$ ;
25  end
26  if  $g_1, g_2, g_3$  are all fork groupings then
27    | return  $[g_1, [[1,1,1], [2,2,2]]]$ ;
28  end
29  // Combine the A-Connections
30  Grouping $\times$ TripleTuple  $[gA, ttA] = \text{TripleProduct}(g_1.AConnection, g_2.AConnection,$ 
31  |  $g_3.AConnection)$ ;
32  InternalGrouping  $g = \text{new InternalGrouping}(g1.level)$ ;
33   $g.AConnection = gA$ ;
34   $g.ARReturnTuple = [1..|ttA|]$ ; // Represents the middle vertices
35   $g.numberOfBConnections = |ttA|$ ;

```

E.3 Existential Quantification

For a CFLOBDD that represents a Boolean function f , existential quantification with respect to the Boolean variable at index i yields $f|_{x_i==T} \vee f|_{x_i==F}$. This operation can be implemented using two calls to Restrict, followed by a final call on BinaryApplyAndReduce to perform the “or.”

Truth Table	Defining Expression	Definition Using ITE
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & F \\ T & F & F \end{array} \end{array} $	$\lambda a, b.F$	$\lambda a, b.F$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & F \\ T & F & T \end{array} \end{array} $	$\lambda a, b.a \wedge b$	$\lambda a, b.\text{ITE}(a, b, F)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & F \\ T & F & F \end{array} \end{array} $	$\lambda a, b.a \wedge \neg b$	$\lambda a, b.\text{ITE}(a, \neg b, F)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & F \\ T & T & T \end{array} \end{array} $	$\lambda a, b.a$	$\lambda a, b.a$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & T \\ T & F & F \end{array} \end{array} $	$\lambda a, b.\neg a \wedge b$	$\lambda a, b.\text{ITE}(a, F, b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & T \\ T & F & T \end{array} \end{array} $	$\lambda a, b.b$	$\lambda a, b.b$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & T \\ T & T & F \end{array} \end{array} $	$\lambda a, b.a \oplus b$	$\lambda a, b.\text{ITE}(a, \neg b, b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{F} & T \\ T & T & T \end{array} \end{array} $	$\lambda a, b.a \vee b$	$\lambda a, b.\text{ITE}(a, T, b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & F \\ T & F & F \end{array} \end{array} $	$\lambda a, b.\neg(a \vee b)$	$\lambda a, b.\text{ITE}(a, F, \neg b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & F \\ T & F & T \end{array} \end{array} $	$\lambda a, b.\neg(a \oplus b)$	$\lambda a, b.\text{ITE}(a, b, \neg b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & F \\ T & T & F \end{array} \end{array} $	$\lambda a, b.\neg b$	$\lambda a, b.\text{ITE}(b, F, T)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & F \\ T & T & T \end{array} \end{array} $	$\lambda a, b.a \vee \neg b$	$\lambda a, b.\text{ITE}(a, T, \neg b)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & T \\ T & F & F \end{array} \end{array} $	$\lambda a, b.\neg a$	$\lambda a, b.\text{ITE}(a, F, T)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & T \\ T & F & T \end{array} \end{array} $	$\lambda a, b.\neg a \vee b$	$\lambda a, b.\text{ITE}(a, b, T)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & T \\ T & T & F \end{array} \end{array} $	$\lambda a, b.\neg(a \wedge b)$	$\lambda a, b.\text{ITE}(a, \neg b, T)$
$ \begin{array}{cc} & b \\ a & \begin{array}{cc cc} & F & T \\ F & \boxed{T} & T \\ T & T & T \end{array} \end{array} $	$\lambda a, b.T$	$\lambda a, b.T$

Algorithm 28: TripleProduct Contd.

```

35   // Combine the B-connections, but only for triples in ttA
36   // Descriptor of triples of exit vertices
37   Tuple ttAns = [];
38   // Create a B-Connection for each middle vertex
39   for j ← 1 to |ttA| do
40     Grouping×TripleTuple [gB,ttB] = TripleProduct(g1.BConnections[ttA(j)(1)],
41                                                 g2.BConnections[ttA(j)(2)], g3.BConnections[ttA(j)(3)]);
42     g.BConnections[j] = gB;
43     g.BReturnTuples[j] = [];
44     for i ← 1 to |ttB| do
45       c1 = g1.BReturnTuples[ttA(j)(1)](ttB(i)(1));
46       c2 = g2.BReturnTuples[ttA(j)(1)](ttB(i)(2));
47       c3 = g3.BReturnTuples[ttA(j)(1)](ttB(i)(3));
48       // Not a new exit vertex of g
49       if [c1,c2,c3] ∈ ttAns then
50         index = the k such that ttAns(k) == [c1,c2,c3];
51         g.BReturnTuples[j] = g.BReturnTuples[j] || index;
52       else
53         g.numberOfExits = g.numberOfExits + 1;
54         g.BReturnTuples[j] = g.BReturnTuples[j] || g.numberOfExits;
55         ttAns = ttAns || [c1,c2,c3];
56       end
57     end
58   end
59   return [RepresentativeGrouping(g), ttAns];
60 end

```

Algorithm 29: RestrictCFLOBDD

Input: CFLOBDD c representing f , int i – index, bool $v - T/F$

Output: CFLOBDD c' representing $f' = f|_{x_i=v}$

```

1 begin
2   Grouping×ReturnTuple [g, rt] = RestrictGrouping(c.grouping, i, v);
3   return RepresentativeCFLOBDD(g, [g.valueTuple[rt[i]] | i ∈ [1..|rt|]]);
4 end

```

F BOOLEAN OPERATIONS VIA ITE

The ternary operation ITE [16] (for “If-Then-Else”), is defined as follows:

$$\text{ITE}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c).$$

Fig. 32 shows how the ternary ITE operation can be used to implement all 16 of the binary operations on Boolean-valued CFLOBDDs [16].

Algorithm 30: RestrictGrouping

Input: Grouping g, int i, bool v
Output: Grouping×ReturnTuple [g', grt]

```

1 begin
2   if g == ForkGrouping then
3     return [DontCareGrouping, (v == False) ? [1] : [2]];
4   end
5   if g == DontCareGrouping || g == NoDistinctionProtoCFLOBDD(g.level) then
6     return [g, [1]];
7   end
8   InternalGrouping g' = new InternalGrouping(g.level);
9   if i < 2*(g.level-1) then // i falls in AConnection range
10    Grouping×ReturnTuple [aa, apt] = RestrictGrouping(g.AConnection, i, v);
11    g'.AConnection = aa;
12    g'.AReturnTuple = [1..apt];
13    g'.numberOfBConnections = |apt|;
14    for j ← 1 to |apt| do
15      g'.BConnections[j] = g.BConnections[apt[j]];
16      // Fill in g'.BReturnTuples[j] from g.BReturnTuples[apt[j]]
17      // appropriately, along with populating grt
18      // Keep track of number of exits of g'
19    end
20   else // i falls in BConnections range
21     for j ← 1 to g.numberOfBConnections do
22       Grouping×ReturnTuple [bb, bpt] = RestrictGrouping(g.BConnections[j], i-(1 «
23         (g.level-1)), v);
24       g'.BConnections[j] = bb;
25       // Fill in g'.BReturnTuples[j] from bpt appropriately, along with
26       // populating grt
27       // Keep track of number of exits of g'
28     end
29     g'.AConnection = g.AConnection;
30     g'.AReturnTuples = [1..|distinct g'.BConnections|];
31     g'.numberOfBConnections = |distinct g'.BConnections|;
32   end
33   g'.numberOfExits = number of exits tracked so far;
34   return [RepresentativeGrouping(g'), grt];
35 end
```

G KRONECKER PRODUCT

This section presents and discusses pseudo-code for the variant of Kronecker product sketched in §7.5.1. Given CFLOBDDs for matrices W and V , with the interleaved-variable orderings $x \bowtie y$

and $w \bowtie z$, respectively, the goal is to create a CFLOBDD for $W \otimes V$ with variable ordering $(x||w) \bowtie (y||z)$.³⁵

Algorithm 31: Kronecker Product

Input: CFLOBDDs n_1, n_2 with variable ordering of $n_1: x \bowtie y$ and $n_2: w \bowtie z$
Output: CFLOBDD $n = n_1 \otimes n_2$ with variable ordering of $n: (x||w) \bowtie (y||z)$

```

1 begin
2   // Create a CFLOBDD of size level( $n_1$ ) + 1 with  $n_1$  as the AConnection
3   CFLOBDD g1 = RepresentativeCFLOBDD(ShiftToAConnection( $n_1$ .grouping),
4      $n_1$ .valueTuple);
5   // Create a CFLOBDD of size level( $n_2$ ) + 1 with  $n_2$  as the BConnection
6   CFLOBDD g2 = RepresentativeCFLOBDD(ShiftToBConnection( $n_2$ .grouping),
7      $n_2$ .valueTuple);
8   CFLOBDD n = BinaryApplyAndReduce(g1, g2, (op)Times);
9   return n;
10
11 end

```

Algorithm 32: ShiftToAConnection

Input: Grouping g
Output: Grouping g' such that AConnection of $g' = g$

```

1 begin
2   InternalGrouping g' = new InternalGrouping( $g$ .level + 1);
3   g'.AConnection = g;
4   g'.AReturnTuple = [1..|g.numberOfExits|];
5   g'.numberOfBConnections = |g.numberOfExits|;
6   for  $j \leftarrow 1$  to  $g$ .numberOfExits do
7     | g'.BConnection[j] = NoDistinctionProtoCFLOBDD( $g$ .level);
8     | g'.BReturnTuples[j] = [j];
9   end
10  g'.numberOfExits = |g.numberOfExits|;
11  return RepresentativeGrouping( $g'$ );
12 end

```

Fig. 13a shows a level- k CFLOBDD for some array W , where W 's value tuple is $[w_0, w_1, w_2]$; Fig. 13b shows a level- k CFLOBDD for some array V , where V 's value tuple is $[v_0, v_1, v_2]$. (W and V could have been embedded into level- $k+1$ CFLOBDDs; for the sake of clarity, we have not depicted such structures.) Fig. 13c shows the level- $k+1$ CFLOBDD that would be constructed by Alg. 31 before any collapsing of the value tuple via Reduce in the call to BinaryApplyAndReduce in line [4].

Under the variable order $(x||w) \bowtie (y||z)$, as we work through the CFLOBDD shown in Fig. 13c for a given assignment, the values of the first 2^k Boolean variables lead us to a middle vertex of the level- $k+1$ grouping. This path will be continued according to the values of the next 2^k variables. Call these two paths p_A and p_B , respectively. Under the interleaved-variable ordering, p_A takes us

³⁵ \bowtie denotes the interleaving of two sequences of variables; $||$ denotes concatenation.

Algorithm 33: ShiftToBConnection

Input: Grouping g
Output: Grouping g' such that BConnection of $g' = g$

```

1 begin
2   InternalGrouping  $g' = \text{new InternalGrouping}(g.\text{level} + 1);$ 
3    $g'.\text{AConnection} = \text{NoDistinctionProtoCFLOBDD}(g.\text{level});$ 
4    $g'.\text{AReturnTuple} = [1];$ 
5    $g'.\text{numberOfBConnections} = 1;$ 
6    $g'.\text{BConnection}[1] = g;$ 
7    $g'.\text{numberOfExits} = |g.\text{numberOfExits}|;$ 
8   return RepresentativeGrouping( $g'$ );
9 end

```

to a particular block of the matrix that Fig. 13c represents, and p_B takes us to a particular element of that block.

The path p_A uses the 2^k variables in $x \bowtie y$, and thus can also be thought of as taking us to an exit vertex e_w that in matrix W is associated with some terminal value w_i . The path p_B uses the 2^k variables in $w \bowtie z$, and thus can be thought of as taking us to an exit vertex e_v that in matrix V is associated with some terminal value v_j . In the CFLOBDD shown in Fig. 13c, the terminal value at the end of the path $p_A||p_B$ is $w_i v_j$. This value is exactly what is required of the matrix $W \otimes V$. After Reduce is called on Fig. 13c, by the canonicity property, the multi-terminal CFLOBDD that results must be the unique representation of $W \otimes V$ under the variable ordering $(x||w) \bowtie (y||z)$.

H EFFICIENT CONSTRUCTION OF Column1Matrix_n

Column1Matrix_n is a square matrix of size $2^n \times 2^n$ in which the first column is filled with 1s, and all other entries are 0.

$$\text{Column1Matrix}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix}_{2^n \times 2^n}$$

Column1Matrix_n can be recursively defined in terms of the matrices $\text{Column1Matrix}_{n/2}$ (of size $2^{n/2} \times 2^{n/2}$) and $O_{n/2}$ (the all-zero matrix of size $2^n \times 2^n$).

$$\text{Column1Matrix}_n = \begin{cases} \begin{bmatrix} \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Column1Matrix}_{n/2} & O_{n/2} & \cdots & O_{n/2} \end{bmatrix}_{2^n \times 2^n} & n > 1 \\ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & n = 1 \end{cases}$$

Base Case. The CFLOBDD representation for the base case of $n = 1$, Column1Matrix_1 , is shown in Fig. 33a. The base-case matrix requires two Boolean variables x_0 and y_0 : x_0 specifies the row,

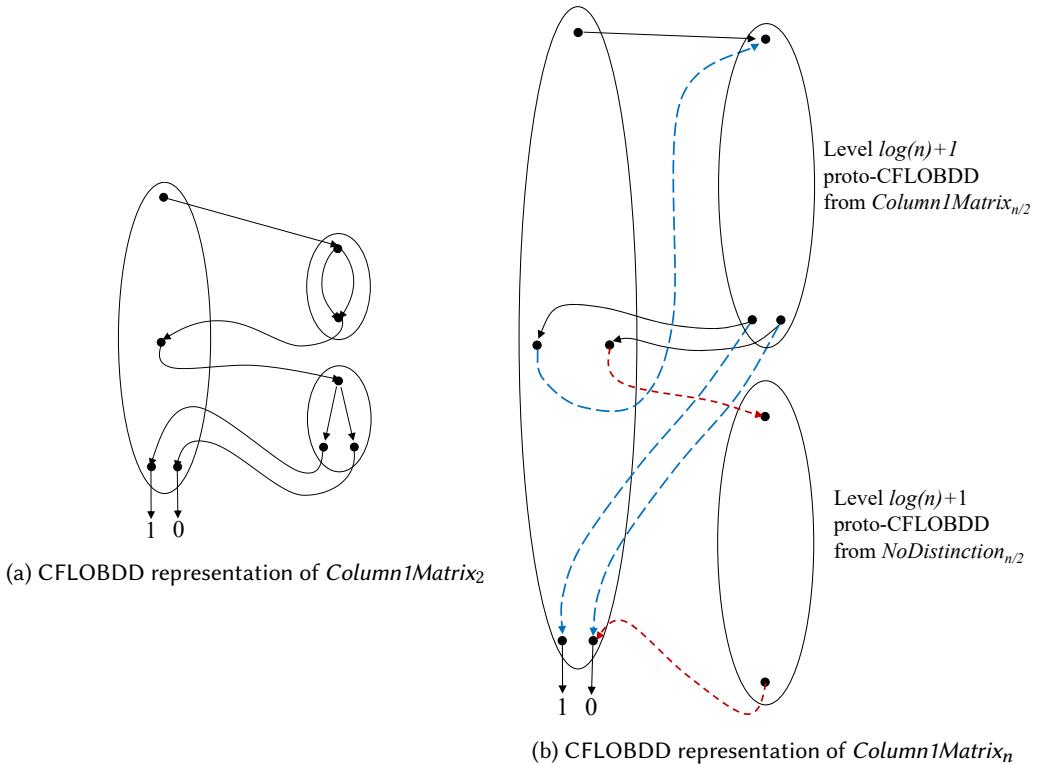


Fig. 33. (a) Base case; (b) the recursive structure for general n .

and y_0 specifies the column; hence, the CFLOBDD that represents $Column1Matrix_1$ has two levels, 0 and 1. The rows of $Column1Matrix_1$ are identical, so the *A-Connection* grouping at level 1 (for x_0) is a *DontCareGrouping*. In contrast, the columns of $Column1Matrix_1$ are not identical, so the *B-Connection* grouping at level 1 (for y_0) is a *ForkGrouping*. See Fig. 33a and lines [5]–[11] of *ColumnMatrixGrouping* in Alg. 34. Note that the left exit vertex of the level-1 proto-CFLOBDD represents the first-column entries of the matrix (which are to have the value 1), and the right exit vertex represents the matrix entries that are to have the value 0.

General Case ($n > 1$). The steps to create $Column1Matrix_n$, for $n > 1$, are shown in lines [13]–[20] of *ColumnMatrixGrouping* in Alg. 34. The number of levels in the CFLOBDD that represents $Column1Matrix_n$ is $\log n + 2$, to provide for the needed $2n$ Boolean variables. (The leaves are at level 0, so the outermost level is $l = \log n + 1$.) The *A-Connection* of the level- l grouping represents a function involving the most-significant $\frac{n}{2}$ row and $\frac{n}{2}$ column variables. From the recursive definition of $Column1Matrix_n = Column1Matrix_{n/2} \otimes Column1Matrix_{n/2}$, the function for the first $\frac{n}{2}$ variables is obtained via a recursive call on $Column1Matrix_n$ with half of the variables. Hence, the *A-Connection* grouping at level- l is a proto-CFLOBDD that represents $Column1Matrix_{n/2}$ (Fig. 33b and line [14] in Alg. 34). The number of exits of this proto-CFLOBDD is two (for which the invariant is maintained that the left exit vertex of the level- $l-1$ proto-CFLOBDD is associated with 1, and the right exit vertex is associated with 0). The grouping at level l therefore has two *B-Connection* groupings, the first one being a second use of the proto-CFLOBDD for $Column1Matrix_{n/2}$ (Fig. 33b

Algorithm 34: Column1Matrix

```

1 Algorithm ColumnMatrixCFLOBDD(l)
    Input: int l - level of the CFLOBDD =  $(\log n) + 1$ , where  $2n$  = the number of variables
    Output: CFLOBDD c representing  $Column1Matrix_n$ 
    begin
        2     Grouping g = Column1MatrixGrouping(l);
        3     return RepresentativeCFLOBDD(g, [1,0]);
    4 end
5 end
6 end
1 SubRoutine ColumnMatrixGrouping(l)
    Input: int l - level of the CFLOBDD =  $(\log n) + 1$ , where  $2n$  = the number of variables
    Output: Grouping g representing proto -  $Column1Matrix_n$ 
    begin
        2     InternalGrouping g = new InternalGrouping(l);
        3 if l == 1 then
            4         g.AConnection = ForkGrouping;
            5         g.AReturnTuples = [1,2];
            6         g.numberOfBConnections = 2;
            7         g.BConnection[1] = DontCareGrouping;
            8         g.ReturnTuples[1] = [1];
            9         g.BConnection[2] = DontCareGrouping;
            10        g.BReturnTuples[2] = [2];
        11 else
            12        Grouping g' = ColumnMatrixGrouping(l-1);
            13        g.AConnection = g';
            14        g.AReturnTuples = [1,2];
            15        g.numberOfBConnections = 2;
            16        g.BConnection[1] = g';
            17        g.BReturnTuples[1] = [1,2];
            18        g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
            19        g.BReturnTuples[2] = [2];
        20 end
        21        g.numberOfExits = 2;
        22        return RepresentativeGrouping(g);
    23 end
24 end
25 end

```

and line [17] in Alg. 34). The second *B-Connection* grouping at level-*l* is a proto-CFLOBDD for $NoDistinction_{n/2}$, representing $O_{n/2}$. This recursive structure is shown in Fig. 33b.

At top level, the level-*l* grouping has two exit vertices—the first maps to the value 1 and the second maps to 0.

Algorithm 35: Controlled-NOT matrix

```

1 Algorithm CNOTCFLOBDD( $n, i, j$ )
  Input: int  $n$ , where  $2n = \#$ Variables of the CFLOBDD, int  $i$  - control-bit; int  $j$  -
    controlled-bit
  Output: CFLOBDD  $c$  representing  $\text{CNOT}(n, i, j)$ 
2 begin
3   Grouping  $g = \text{CNOTGrouping}(\log n + 1, i, j)$ ;
4   return RepresentativeCFLOBDD( $g, [1,0]$ );
5 end
6 end
1 SubRoutine CNOTGrouping( $l, i, j$ )
  Input: int  $l$  - grouping level, int  $i$  - control-bit; int  $j$  - controlled-bit
  Output: Grouping  $g$  representing  $\text{CNOT}(l, i, j)$  with  $n = 2^l$  bits
2 begin
3   if  $l == 2$  then // Base Case
4     InternalGrouping  $g = \text{new InternalGrouping}(2)$ ;
5     InternalGrouping  $g' = \text{new InternalGrouping}(1)$ ; // Level 1
6      $g.\text{AConnection} = \text{ForkGrouping}$ ;
7      $g'.\text{AReturnTuple} = [1,2]$ ;
8      $g'.\text{numberOfBConnections} = 2$ ;
9      $g'.\text{BConnection}[1] = \text{ForkGrouping}$ ;
10     $g'.\text{BReturnTuples}[1] = [1,2]$ ;
11     $g'.\text{BConnection}[2] = \text{ForkGrouping}$ ;
12     $g'.\text{BReturnTuples}[2] = [2,3]$ ;
13     $g'.\text{numberOfExits} = 3$ ;
14     $g.\text{AConnection} = g'$ ;
15     $g.\text{AReturnTuple} = [1,2,3]$ ;
16     $g.\text{numberOfBConnections} = 3$ ;
17     $g.\text{BConnection}[1] = \text{IdentityMatrixGrouping}(1)$ ;
18     $g.\text{BReturnTuples}[1] = [1,2]$ ;
19     $g.\text{BConnection}[2] = \text{NoDistinctionProtoCFLOBDD}(1)$ ;
20     $g.\text{BReturnTuples}[2] = [2]$ ;
21     $g.\text{BConnection}[3] = \text{IdentityMatrixGrouping}(1)$ ;
22     $g.\text{BReturnTuples}[3] = [2,1]$ ;
23     $g.\text{numberOfExits} = 2$ ;
24    return RepresentativeGrouping( $g$ );
25 end

```

I ALGORITHM FOR CONSTRUCTING THE CNOT MATRIX

Pseudo-code for the algorithm for constructing the CFLOBDD that represents the Controlled-NOT matrix is given in Algs. 35–38. The algorithm for the special-case construction of CNOT_n discussed in §9.2.5 is given in Alg. 39.

Algorithm 36: CNOT contd.

```

27
28
29   if i and j fall in A-connection range then // Case 1
30     InternalGrouping g = new InternalGrouping(l + 1);
31     g.AConnection = CNOTGrouping(l-1, i, j);
32     g.AReturnTuple = [1,2];
33     g.numberOfBConnections = 2;
34     g.BConnection[1] = IdentityMatrixGrouping(g.level - 1);
35     g.BReturnTuples[1] = [1,2];
36     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
37     g.BReturnTuples[2] = [2];
38     g.numberOfExits = 2;
39     return RepresentativeGrouping(g);
40
41   end
42   if i and j fall in B-connection range then // Case 2
43     InternalGrouping g = new InternalGrouping(l + 1);
44     g.AConnection = IdentityMatrixGrouping(g.level - 1);
45     g.AReturnTuple = [1,2];
46     g.numberOfBConnections = 2;
47     g.BConnection[1] = CNOTGrouping(l-1, i', j'); // i' = i - 2l-1, j' = j - 2l-1
48     g.BReturnTuples[1] = [1,2];
49     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
50     g.BReturnTuples[2] = [2];
51     g.numberOfExits = 2;
52     return RepresentativeGrouping(g);
53
54   end
55   if i in A-connection range and j in B-connection range then // Case 3
56     InternalGrouping g = new InternalGrouping(l + 1);
57     g.AConnection = CNOTGrouping(l-1, i, -1);
58     g.AReturnTuple = [1,2,3];
59     g.numberOfBConnections = 3;
60     g.BConnection[1] = IdentityMatrixGrouping(g.level-1);
61     g.BReturnTuples[1] = [1,2];
62     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
63     g.BReturnTuples[2] = [2];
64     g.BConnection[3] = CNOTGrouping(l-1, -1, j'); // j' = j - 2l-1
65     g.BReturnTuples[3] = [2,1];
66     g.numberOfExits = 2;
67     return RepresentativeGrouping(g);
68
69   end

```

Algorithm 37: CNOT contd.

```

68
69
70   if i in A-connection range but j is not in this range then           // Case 4
71     InternalGrouping g = new InternalGrouping(l + 1);
72     g.AConnection = CNOTGrouping(l-1, i, -1);
73     g.AReturnTuple = [1,2,3];
74     g.numberOfBConnections = 3;
75     g.BConnection[1] = IdentityMatrixGrouping(g.level-1);
76     g.BReturnTuples[1] = [1,2];
77     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
78     g.BReturnTuples[2] = [2];
79     g.BConnection[3] = IdentityMatrixGrouping(g.level-1);
80     g.BReturnTuples[3] = [3,2] ;
81     g.numberOfExits = 3;
82     return RepresentativeGrouping(g);
83   end
84   if i in B-connection range but j is not in this range then           // Case 5
85     InternalGrouping g = new InternalGrouping(l + 1);
86     g.AConnection = IdentityMatrixGrouping(g.level-1);
87     g.AReturnTuple = [1,2];
88     g.numberOfBConnections = 2;
89     g.BConnection[1] = CNOTGrouping(l-1, i', -1);                  // i' = i - 2l-1
90     g.BReturnTuples[1] = [1,2,3];
91     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
92     g.BReturnTuples[2] = [2];
93     g.numberOfExits = 3;
94     return RepresentativeGrouping(g);
95   end
96   if j in A-connection range but i is not in this range then           // Case 6
97     InternalGrouping g = new InternalGrouping(l + 1);
98     g.AConnection = CNOTGrouping(l-1, -1, j);
99     g.AReturnTuple = [1,2];
100    g.numberOfBConnections = 2;
101    g.BConnection[1] = NoDistinctionProtoCFLOBDD(g.level-1);
102    g.BReturnTuples[1] = [1,2];
103    g.BConnection[2] = IdentityMatrixGrouping(g.level - 1);
104    g.BReturnTuples[2] = [1];
105    g.numberOfExits = 2;
106    return RepresentativeGrouping(g);
107  end

```

J CONSTRUCTION OF ADDITIONAL QUANTUM GATES

In this section, we discuss the construction of two quantum gates: the Controlled Phase gate and the Swap gate.

Algorithm 38: CNOT contd.

```

109
110
111   if  $j$  in B-connection range but  $i$  is not in this range then           // Case 7
112     InternalGrouping g = new InternalGrouping( $l + 1$ );
113     g.AConnection = IdentityMatrixGrouping(g.level-1);
114     g.ARearTuple = [1,2];
115     g.numberOfBConnections = 2;
116     g.BConnection[1] = CNOTGrouping(l-1, -1,  $j'$ );                  //  $j' = j - 2^{l-1}$ 
117     g.BReturnTuples[1] = [1,2];
118     g.BConnection[2] = NoDistinctionProtoCFLOBDD(g.level - 1);
119     g.BReturnTuples[2] = [1];
120     g.numberOfExits = 2;
121     return RepresentativeGrouping(g);
122   end
123 end
124 end

```

J.1 Controlled-Phase Gate

A Controlled-Phase gate (CP) for two qubits with angle θ has the following matrix:

$$CP(\theta) = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 00 & 1 & 0 & 0 \\ 01 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 \\ 11 & 0 & 0 & 0 \\ & & & e^{i\theta} \end{bmatrix}$$

More generally, the controlled-phase gate with control qubit i and controlled qubit j adds the phase angle θ to the j^{th} qubit when both qubits have the value 1. The construction of the CFLOBDD that represents a CP gate's matrix for n qubits is similar to the construction of the CFLOBDD for the $CNOT$ matrix. We do not give pseudo-code, but Fig. 34 depicts the different cases of the construction of the proto-CFLOBDD for CP . The CFLOBDD for a CP gate attaches the three terminal values $[1, 0, e^{i\theta}]$.

CP takes 4 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, where $0 \leq i < j < n$; and the phase θ . The key to understanding Fig. 34 is that the construction maintains the invariant shown in Eqn. (37) on the exit vertices of the different kinds of groupings.

	Role	Significance of exit vertex		
		1	2	3
Proto-CFLOBDD	$CP(n, i, j)$	on-path	off-path	phase-path
	$CP(n, i, \blacksquare)$	on-path	off-path	phase-path
	$CP(n, \blacksquare, j)$	on-path	off-path	phase-path
	ID called from middle vertex 1	on-path	off-path	N/A
$CFLOBDD$	ID called from middle vertex 3	phase-path	off-path	N/A
	Top level	1	0	$e^{i\theta}$

Algorithm 39: Algorithm for constructing $CNOT_n$

```

1 Algorithm CNOTInterleavedMatrixxCFLOBDD(l)
    Input: int l - level of the CFLOBDD =  $\log 2n + 1$ , where  $2n$  = number of bits
    Output: The CFLOBDD that represents  $CNOT_n$ 
2   begin
3     | Grouping g = CNOTInterleavedMatrixGrouping(l);
4     |   return RepresentativeCFLOBDD(g, [1,0]);
5   end
6 end
1 SubRoutine CNOTInterleavedMatrixGrouping(l)
    Input: int l - level of the CFLOBDD =  $\log n$ , where  $2n$  = number of bits
    Output: Grouping g representing the proto-CFLOBDD for  $CNOT_n$ 
2   begin
3     InternalGrouping g = new InternalGrouping(l);
4     if l == 2 then
5       |   return CNOTGrouping(2, 1, 2);                                // The base case is  $CNOT_2$ 
6     else
7       |   Grouping g' = CNOTInterleavedMatrixGrouping(l-1);
8       |   g.AConnection = g';
9       |   g.AReturnTuple = [1,2];
10      |   g.numberOfBConnections = 2;
11      |   g.BConnection[1] = g';
12      |   g.BReturnTuples[1] = [1,2];
13      |   g.BConnection[2] = NoDistinctionProtoCFLOBDD(l-1);
14      |   g.BReturnTuples[2] = [2];
15     end
16     g.numberOfExits = 2;
17     return RepresentativeGrouping(g);
18   end
19 end

```

Here, “on-path” means that the exit occurs on a matched-path that can be continued to the top-level terminal value 1; “off-path” means that it will only be used to reach the top-level terminal value 0; and “phase-path” means that the exit occurs on a matched-path that can be continued by IdentityMatrixGroupings to the top-level terminal values θ and 0.

J.2 Swap Gate

A Swap gate is a matrix that swaps two bits (or variables). The Swap matrix for 2 bits (i.e., for 2 input variables and 2 output variables) is

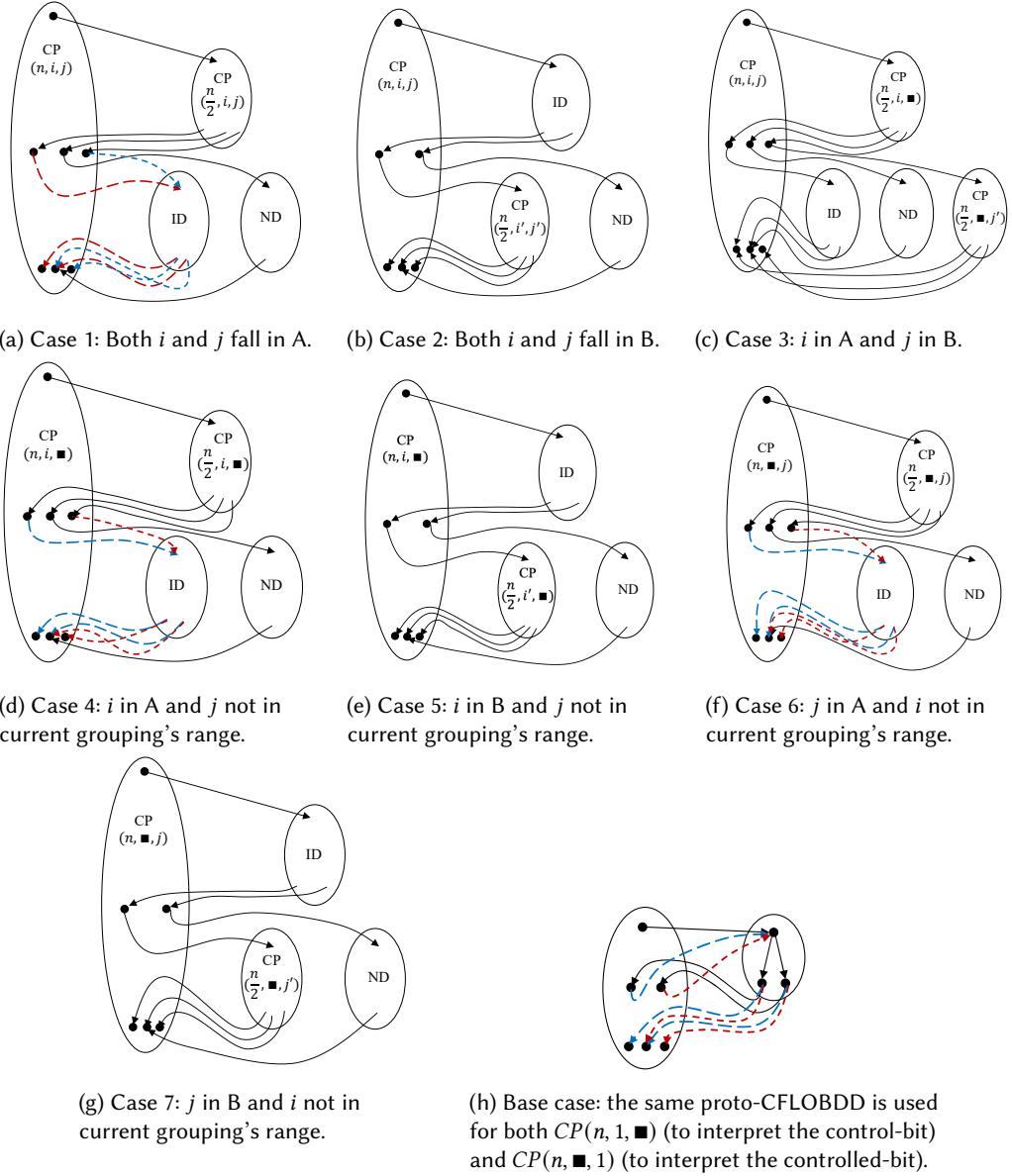


Fig. 34. The different cases of the CP construction. The text in each grouping denotes the function represented by the grouping. ID denotes IdentityMatrixGrouping, and ND denotes a NoDistinctionProtoCFLOBDD (used here for an all-zero matrix). CP takes 4 arguments: n for the number of bits in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, where $0 \leq i < j < n$; and θ (which is only used at top level in the CFLOBDD's value tuple). i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range. Figure (h) shows the base case at level 1; the same proto-CFLOBDD is used for both $CP(n, 1, ■)$ and $CP(n, ■, 1)$.

$$SwapGate = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 00 & 1 & 0 & 0 \\ 01 & 0 & 0 & 1 \\ 10 & 0 & 1 & 0 \\ 11 & 0 & 0 & 1 \end{bmatrix} \quad (38)$$

A matrix entry is 1 in exactly the positions where swapping the bits of the row index yields the bits of the column index. The matrix can be divided into four quadrants that have different values:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \text{ and } \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

This observation comes in handy in the construction of the generalized Swap matrix for higher numbers of bits n .

The construction of the *Swap* matrix is similar to the construction of the *CNOT* matrix. Again, we do not give pseudo-code, but give a graphical depiction of the different cases of the construction. The CFLOBDD for a *Swap* gate always has $[1, 0]$ as the value tuple for the terminal values. Figs. 35, 36, and 37 depict the different cases involved in the construction of the proto-CFLOBDD for *Swap*, given n and the bits i and j to swap (where $i < j$). The construction also uses an additional parameter, called “*State*,” which takes the values $\{0..4\}$. *State* is initially 0, and remains 0 until the Boolean variable corresponding to bit i is encountered. At this point, the construction creates representations of sub-matrices, where *State* takes the values 1, 2, 3, and 4, corresponding to the different 2×2 sub-matrices discussed above. This situation is depicted in the base case shown in Fig. 37d. These states are propagated further through the proto-CFLOBDD (see Fig. 35e, Fig. 35f, Fig. 36a, Fig. 36b, Fig. 37b, and Fig. 37c) until the base cases that interpret the controlled-bit are encountered (see Fig. 37e, Fig. 37f, Fig. 37g, and Fig. 37h).

The key to understanding Figs. 35, 36, and 37 is that the construction maintains the invariant shown in Eqn. (39) on the exit vertices of the different kinds of groupings. (“On-path” means that the exit occurs on a matched-path that can be continued to the top-level terminal value 1; “off-path” means that it will only be used to reach the top-level terminal value 0; and “cd-bit” abbreviates “controlled-bit.”)

	Role	Significance of exit vertex				
		1	2	3	4	5
Proto-CFLOBDD	$SWAP(n, i, j)$	on-path	off-path	N/A	N/A	N/A
	$SWAP(n, i, \blacksquare), cd-bit = \frac{n}{2}-1$	$State=1$	$State=2$	$State=3$	$State=4$	off-path
	$SWAP(n, i, \blacksquare), cd-bit \neq \frac{n}{2}-1$	$State=1$	off-path	$State=2$	$State=3$	$State=4$
	$SWAP(n, \blacksquare, j), State=1$	on-path	off-path	N/A	N/A	N/A
	$SWAP(n, \blacksquare, j), State=2$	off-path	on-path	N/A	N/A	N/A
	$SWAP(n, \blacksquare, j), State=3$	off-path	on-path	N/A	N/A	N/A
	$SWAP(n, \blacksquare, j), State=4$	off-path	on-path	N/A	N/A	N/A
	ID , called from $State=1$	$State=1$	off-path	N/A	N/A	N/A
	ID , called from $State=2$	$State=2$	off-path	N/A	N/A	N/A
	ID , called from $State=3$	$State=3$	off-path	N/A	N/A	N/A
	ID , called from $State=4$	$State=4$	off-path	N/A	N/A	N/A
CFLOBDD	Top level	1	0	N/A	N/A	N/A

(39)

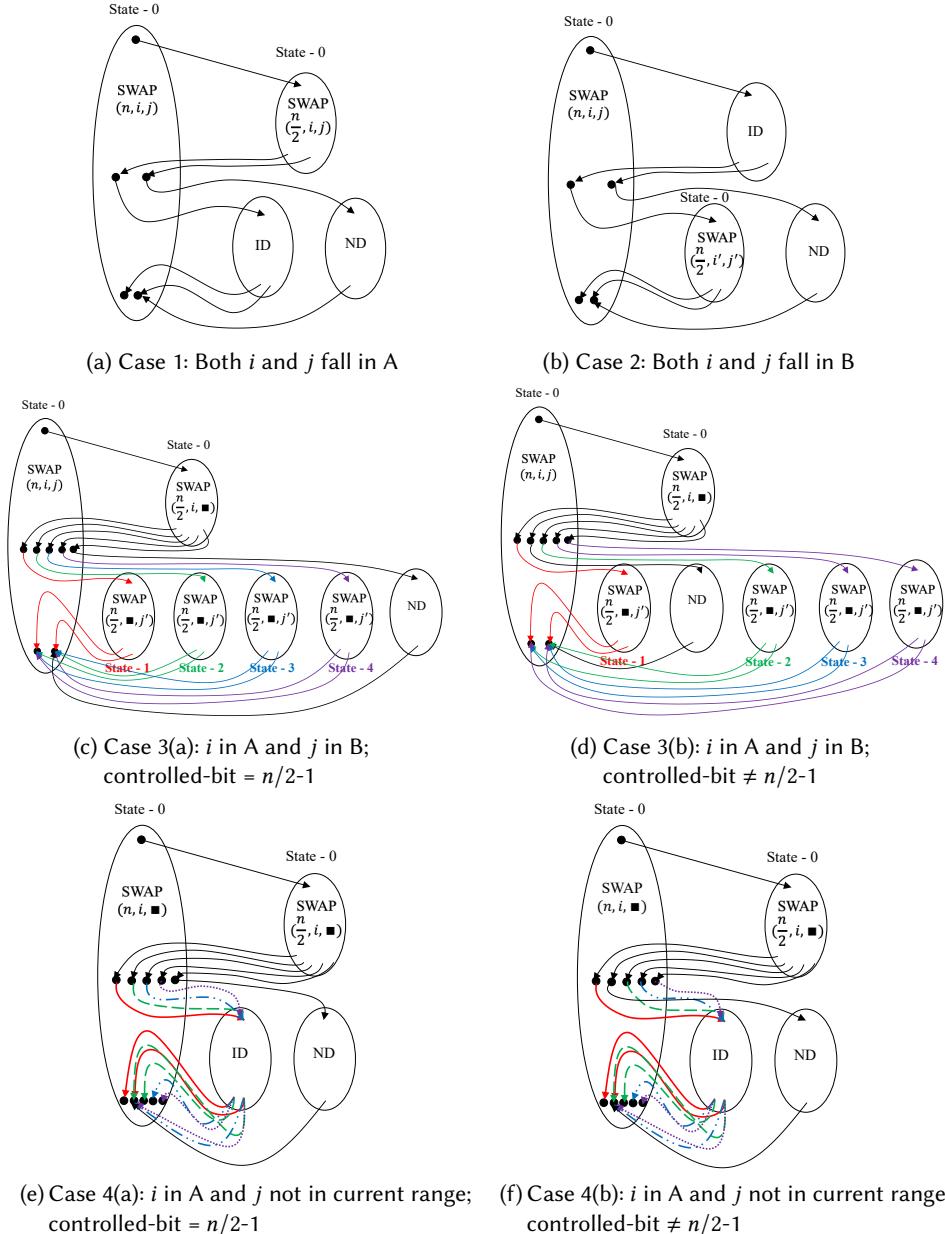


Fig. 35. The different cases of the *SWAP* matrix construction. The text in each grouping denotes the function represented by the grouping. ID denotes *IdentityMatrixGrouping*, and ND denotes a *NoDistinctionProtocFLOBDD* (used here for an all-zero matrix). *SWAP* takes 4 arguments: n for the number of bits (number of variables will be $2n$) in this proto-CFLOBDD; i for the control-bit, j for the controlled-bit, and $State \in \{0, 1, 2, 3, 4\}$ to indicate the current mode of the construction. i' and j' denote bit indices adjusted to the index range of the current level: $i' = i - n/2$; $j' = j - n/2$. A black square indicates that a particular index is outside the grouping's index range.

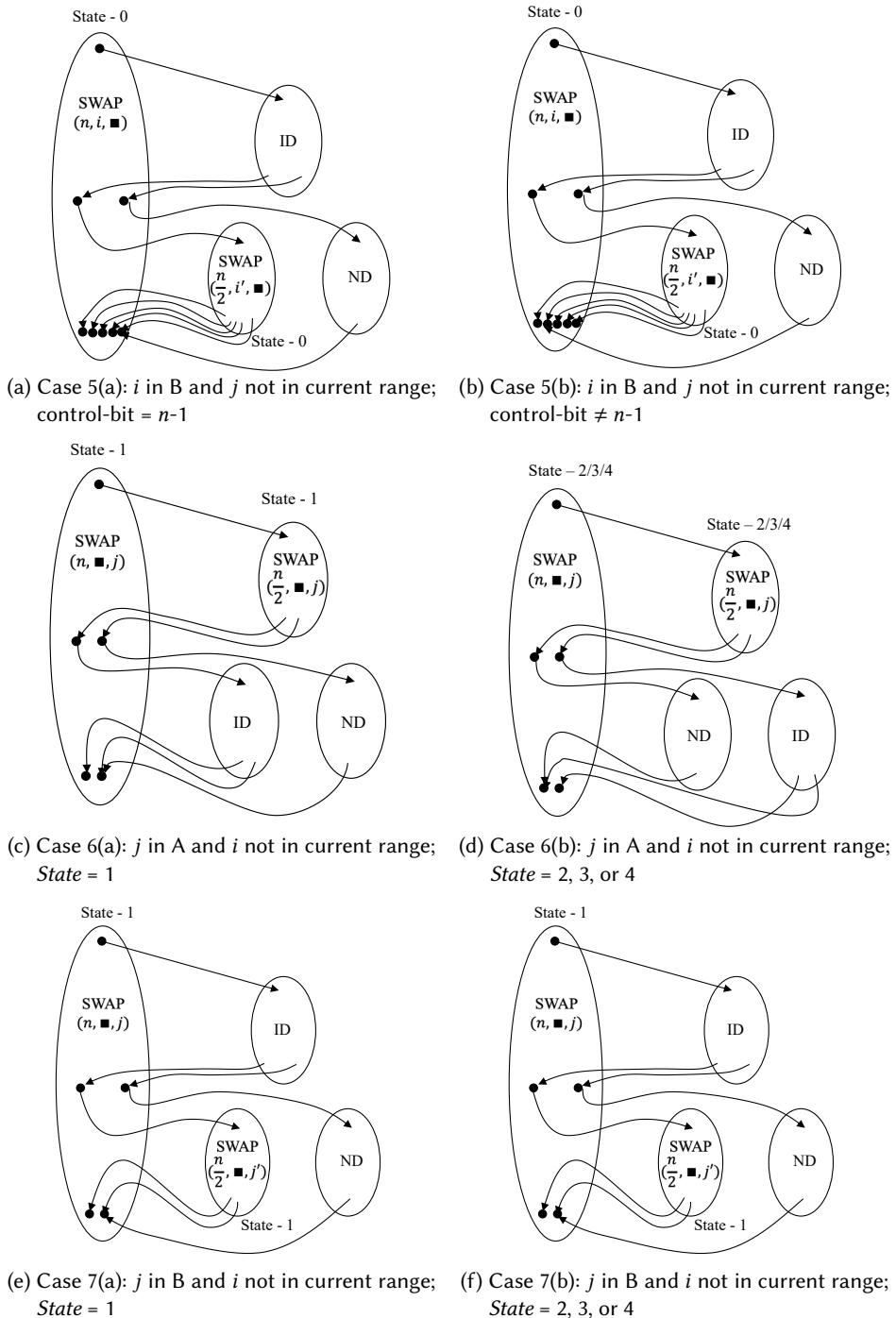


Fig. 36. The different cases of the SWAP matrix construction, continued.

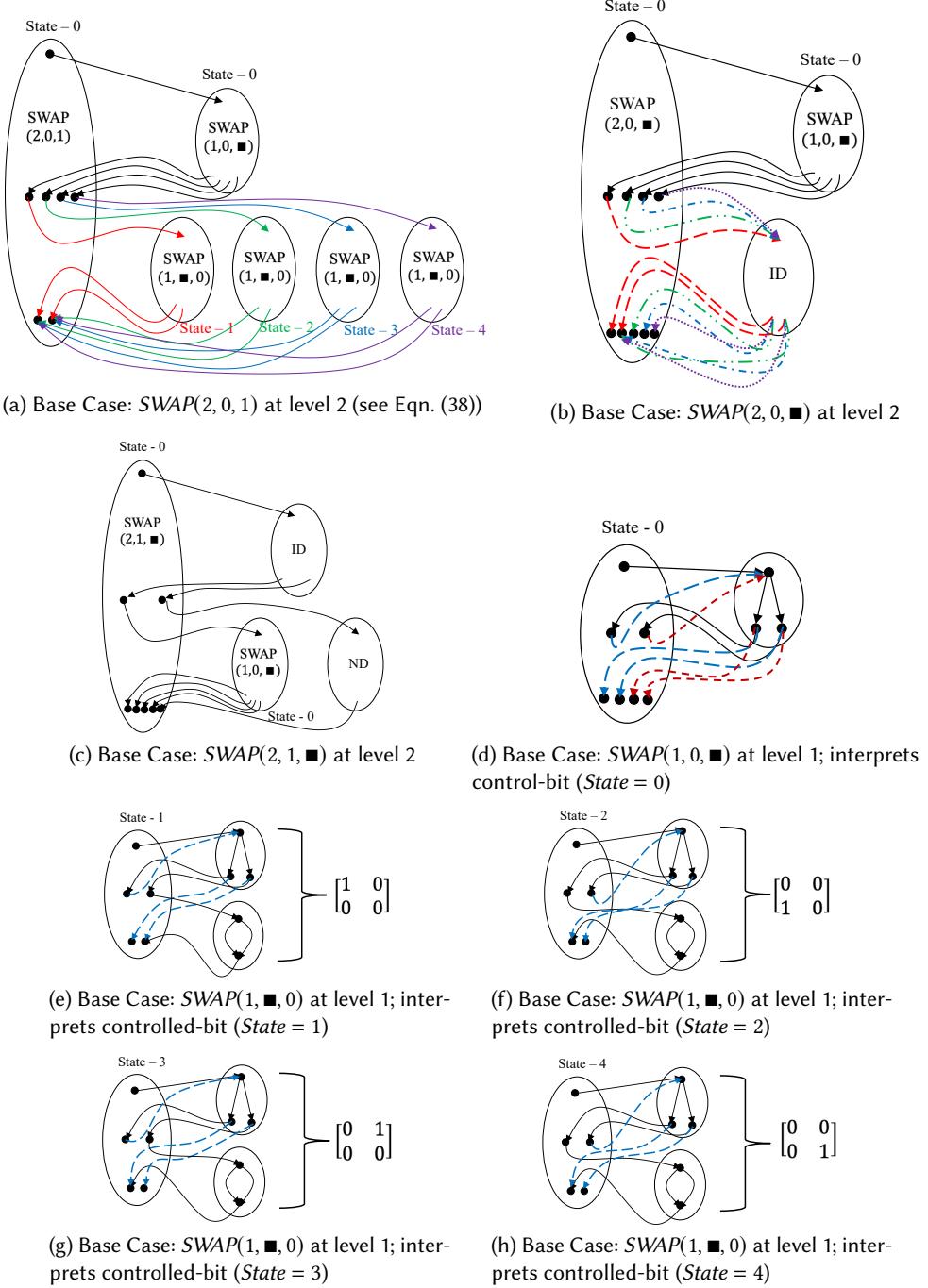
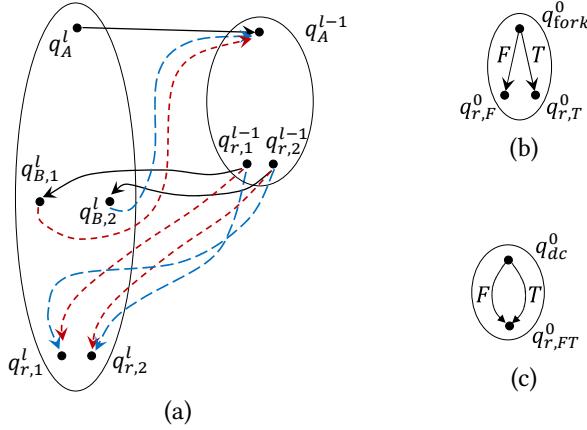


Fig. 37. Base cases for the construction of the SWAP matrix. There are three base cases at level 2 with 2 bits (i.e., 4 Boolean variables) and five base cases at level 1 with 1 bit (i.e., 2 Boolean variables).



		$(q_A^l, \epsilon, q_A^{l-1}) \in \delta_c$
	call transitions	$(q_{B,1}^l, \epsilon, q_A^{l-1}) \in \delta_c$
		$(q_{B,2}^l, \epsilon, q_A^{l-1}) \in \delta_c$
(a)	return transitions	$(q_{r,1}^{l-1}, q_A^l, \epsilon, q_{B,1}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_A^l, \epsilon, q_{B,2}^l) \in \delta_r$
	internal transitions	$(q_{r,1}^{l-1}, q_{B,1}^l, \epsilon, q_{r,1}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_{B,1}^l, \epsilon, q_{r,2}^l) \in \delta_r$
	internal transitions	$(q_{r,1}^{l-1}, q_{B,2}^l, \epsilon, q_{r,2}^l) \in \delta_r$
		$(q_{r,2}^{l-1}, q_{B,2}^l, \epsilon, q_{r,1}^l) \in \delta_r$
(b)	internal transitions	$(q_{fork}^0, F, q_{r,F}^0) \in \delta_i$
		$(q_{fork}^0, T, q_{r,T}^0) \in \delta_i$
(c)	internal transitions	$(q_{dc}^0, F, q_{r,FT}^0) \in \delta_i$
		$(q_{dc}^0, T, q_{r,FT}^0) \in \delta_i$

Fig. 38. (a) Encoding of a grouping's A-connection and B-connections as call transitions, and its return edges as return transitions of an NWA. The grouping is the one used to encode the family of Hadamard matrices \mathcal{H} . (b) and (c) Encoding of the two kinds of level-0 groupings as internal transitions of an NWA.

K NESTED WORDS AND NESTED WORD AUTOMATA

Definition K.1 ([4]). A **nested word** (w, \rightsquigarrow) over alphabet Σ is an ordinary word $w \in \Sigma^*$, together with a **nesting relation** \rightsquigarrow of length $|w|$. \rightsquigarrow is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \dots, l\} \times \{1, 2, \dots, l, +\infty\}$ such that

- Nesting edges only go forwards: if $i \rightsquigarrow j$ then $i < j$.
- No two edges share a position: for $1 \leq i \leq l$, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$.
- Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.

When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, i is called a **call** position; if $i \rightsquigarrow +\infty$, then i is a **pending call**; otherwise i is a **matched call**, and the unique position j such that $i \rightsquigarrow j$ is called its **return successor**. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, j is a **return** position; if $-\infty \rightsquigarrow j$, then j is a **pending return**, otherwise j is a **matched return**, and the unique position i such that $i \rightsquigarrow j$ is

called its **call predecessor**. A position $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.

MatchedNW denotes the set of nested words that have no pending calls or returns. **NWPrefix** denotes the set of nested words that have no pending returns.

A **nested word automaton** (NWA) A is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where

- $\delta_i \subseteq Q \times \Sigma \times Q$ is the transition relation for internal positions.
- $\delta_c \subseteq Q \times \Sigma \times Q$ is the transition relation for call positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation for return positions.

Starting from q_0 , an NWA A reads a nested word $nw = (w, \rightsquigarrow)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and \rightsquigarrow . If A is in state q when reading input symbol σ at position i in w , and i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. If i is a return position, let k be the call predecessor of i , and q_c be the state A was in just before the transition it made on the k^{th} symbol; A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q' . If, after reading nw , A is in a state $q \in F$, then A **accepts** nw .

Fig. 38 illustrates a schema by which a CFLOBDD can be translated to an NWA M . Each matched path through the CFLOBDD corresponds to a nested word in MatchedNW for M . The matched-path principle is obeyed because of the ability of an NWA to “peek” at the state of the most-recent “call” to match a return edge with the appropriate preceding A-connection or B-connection. All transitions taken at a level ≥ 1 are ϵ -transitions (Fig. 38a). The only transitions that consume an alphabet symbol are the F and T transitions of the level-0 fork grouping (Fig. 38b) and the F and T transitions of the level-0 don’t-care grouping (Fig. 38c).

Fig. 39 shows the nested word that corresponds to the path for the assignment $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$ in the CFLOBDD for the Hadamard matrix H_4 , with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$.

L TIME COMPLEXITY OF REDUCE

In this section, we give a bound on the time complexity of the call on Reduce (Alg. 13) in line [5] of BinaryApplyAndReduce (Alg. 10). Let C be the level- l proto-CFLOBDD on which Reduce is invoked, and C' be the level- l proto-CFLOBDD that is returned. The accounting is somewhat subtle because of three factors

- hash-consing of groupings
- function caching of calls to Reduce and other functions
- for $C' = \text{Reduce}(C, \text{red})$ (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.³⁶

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 $O(1)$.

³⁶ 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 $|C|$.

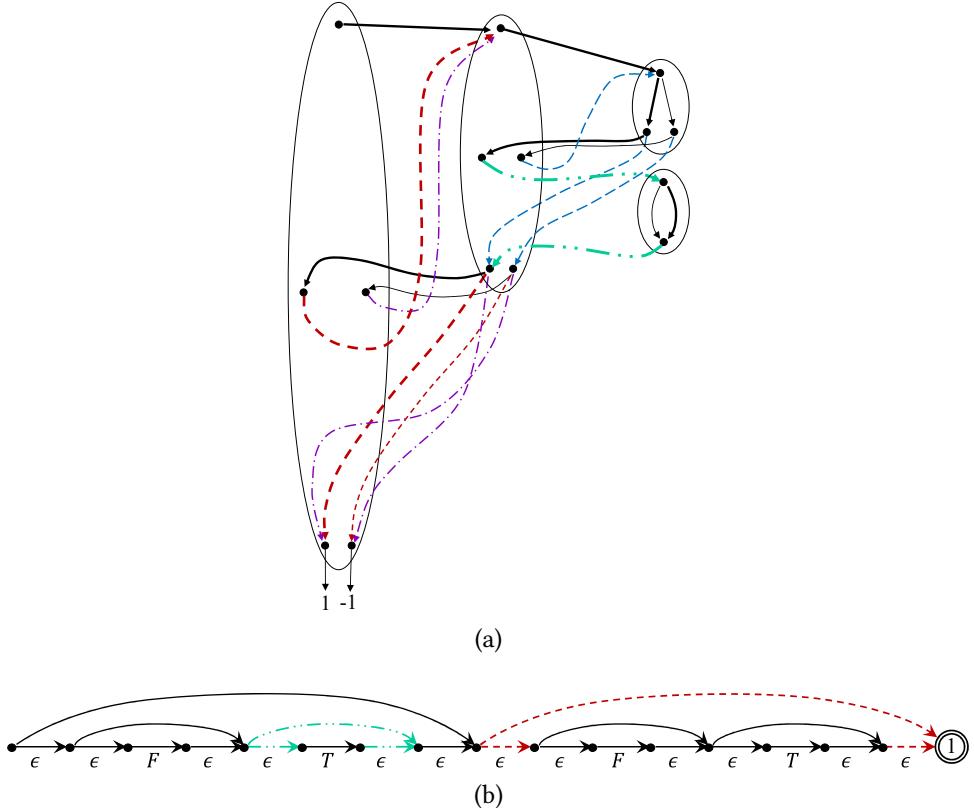


Fig. 39. (a) The CFLOBDD for the Hadamard matrix H_4 , with the variable ordering $\langle x_0, y_0, x_1, y_1 \rangle$ (repeated from Fig. 6a). The matched path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$, which corresponds to $H_4[0, 3]$ (with value 1), is shown in bold. (b) The nested word for the path for $[x_0 \mapsto F, y_0 \mapsto T, x_1 \mapsto F, y_1 \mapsto T]$.

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 L.1. Consider the level-2 proto-CFLOBDD C shown in Fig. 40a, 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 $P_1 = [\{00\}, \{01\}, \{10\}, \{11\}]$, i.e., C_A has four exit vertices and thus C has four middle vertices and four B-connections ($C_{B1}, C_{B2}, C_{B3}, C_{B4}$). C_{B1} partitions the strings $\{0, 1\}^2$ into $P_2 = [\{00\}, \{01, 10\}, \{11\}]$ and its exit vertices are connected to the exit vertices of C in the order (a, b, c) . C_{B2} and C_{B4} both equal C_{B1} , but for C_{B2} and C_{B4} the three exit vertices are connected to the exit vertices of C in the orders (b, d, e) , and (g, a, h) , respectively. Finally, C_{B3} equals C_A , but for C_{B3} the four exit vertices are connected to C 's exit vertices in the order (b, d, e, f) . Consequently, C partitions the strings $\{0, 1\}^4$ into $[\{0000, 1101, 1110\}, \{0001, 0010, 0100, 1000\}, \{0011, 0110, 1001\}, \{0111, 1010\}, \{1011\}, \{1100\}, \{1111\}]$.

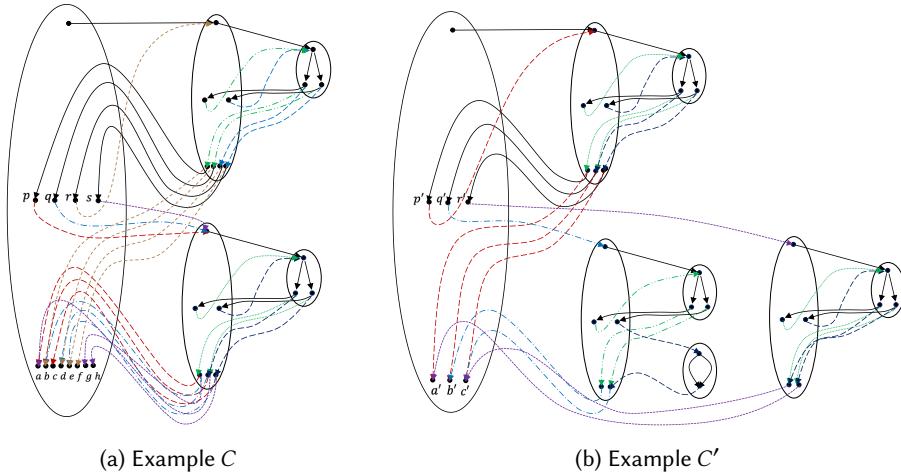


Fig. 40. $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$. The colors of the edges to proto-CFLOBDDs in C' correspond to the edges to the originating proto-CFLOBDDs in C .

Let $C' = \text{Reduce}(C, [1, 2, 3, 3, 3, 3, 3, 3])$, i.e., the vertices c, d, e, f, g, h are all mapped to exit vertex c . C' is shown in Fig. 40b. 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 below, C'_{B1} is also equal to C'_A , the A-connection of C' .)
- C_{B2} ’s exit vertices are mapped to (b, c, c) , which leads to the call $\text{Reduce}(C_{B2}, [1, 2, 2])$. 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 $C' = \text{Reduce}(C, \text{red})$ 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. \square

Because of the effects illustrated in Ex. L.1, the cost-bound argument we give is slightly indirect. At a high-level, it is structured as follows: we establish a relationship between $\text{Reduce}(C, \text{red})$ and that of a certain call on PairProduct (Thm. L.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). Thm. L.3 uses that bound to give an asymptotic bound on the time to perform $\text{Reduce}(C, \text{red})$ in terms of the product of the sizes of its input and output CFLOBDDs.

THEOREM L.1. *Let C and C' be two proto-CFLOBDDs such that $C' = \text{Reduce}(C, \text{red})$ for some reduction tuple red . Then $C = \text{PairProduct}(C, C')$.³⁷*

Proof: We know that each proto-CFLOBDD at level k with m exit vertices partitions the space of strings $\{0, 1\}^{2^k}$ into m groups (see §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 , $\text{Reduce}(X, \text{red})$ 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 , $\text{PairProduct}(X, Y)$ produces the coarsest partition that refines both of the partitions corresponding to X and Y .

In particular, we consider the two-statement sequence

$$C' = \text{Reduce}(C, \text{red}); \quad (40)$$

$$\tilde{C} = \text{PairProduct}(C, C'); \quad (41)$$

C' created in Eqn. (40) represents a coarser partition of the strings in $\{0, 1\}^n$ than C 's partition. Because C' represents a coarser partition than C , the proto-CFLOBDD \tilde{C} created in Eqn. (40) represents the same partition as C , and thus \tilde{C} and C are equal by canonicity. \square

In essence, Thm. L.1 shows that $\text{PairProduct}(C, C')$ “undoes” all of the actions taken during $\text{Reduce}(C, \text{red})$.

EXAMPLE L.2. Consider the result $\tilde{C} = \text{PairProduct}(C, C')$ for C and C' from Ex. L.1. $\text{PairProduct}(C, C')$ is first called on the A-connections of the respective outermost groupings, followed by calls on B-connections.

- $\text{PairProduct}(C_A, C'_A)$ 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

³⁷To reduce clutter, we ignore the tuple of pairs of exit vertices that is returned by PairProduct (Alg. 11), except for two places in Thm. L.3.

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 $\{[p, p'], [q, q'], [r, r'], [s, s']\}$. These pairs are processed left-to-right, generating calls to PairProduct on B-connections.

- $\text{PairProduct}(C_{B1}, C'_{B1})$ (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 $\{[a, a'], [b, b'], [c, c']\}$.
- $\text{PairProduct}(C_{B2}, C'_{B2})$ (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 $[e, c']$.
- $\text{PairProduct}(C_{B3}, C'_{B2})$ (corresponding to the pair $[r, r']$) 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 $[f, c']$.
- $\text{PairProduct}(C_{B4}, C'_{B3})$ (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, $\text{PairProduct}(C, C')$ would return a pointer to C . \square

LEMMA L.2. (Local-Reduction Property). *Let C and C' be two proto-CFLOBDDs such that $C' = \text{Reduce}(C, \text{red})$ for some reduction tuple red , and let g and g' be their respective outermost groupings. Then $|g'| \leq |g|$.*

Proof: The size of a grouping is equal to the number of entry, middle, and exit vertices, plus the number of A-connection and B-connection edges and return edges. Because g' is obtained by reducing g with respect to red , the number of exit vertices in g' can be no more than the number in g . Moreover, Reduce can never cause there to be more B-connections in g' than in g , but it can cause some B-connections of g to be folded together in g' ; thus, the number of middle vertices in g' can be no more than the number in g . Similarly, for the A-connection of g' and all the B-connections of g' , the number of return edges can be no more than the number of return edges in the corresponding A-/B-connections in g . Consequently, $|g'| \leq |g|$. \square

EXAMPLE L.3. Consider the proto-CFLOBDDs C and C' from Fig. 40. The size of the level-2 grouping g equals 1 (entry-vertex) + 4 (middle vertices) + (1 + 3) (1st B-connection) + (1 + 3)(2nd B-connection) + (1 + 4) (3rd B-connection) + (1 + 3)(4th B-connection) + 8 (exit vertices) = 30.

The size of g' equals 1 (entry-vertex) + 3 (middle vertices) + (1 + 3) (1st B-connection) + (1 + 2)(2nd B-connection) + (1 + 2) (3rd B-connection) + 3 (exit vertices) = 17.

Thus, $|g'| \leq |g|$, whereas $68 = |C'| > |C| = 66$. \square

We now turn to the question of bounding the time complexity of Reduce. Whereas Thm. L.1 showed that $\text{PairProduct}(C, C')$ “undoes” all of the actions taken during $\text{Reduce}(C, \text{red})$, Thm. L.3 shows that for every action in $\text{Reduce}(C, \text{red})$, there is an action of at least the same cost in $\text{PairProduct}(C, C')$. Consequently, the time to perform $\text{Reduce}(C)$ is bounded by the time that it would take to perform $\text{PairProduct}(C, C')$, which is $O(|C| \times |C'|)$.

THEOREM L.3. *Let C and C' be two proto-CFLOBDDs such that $C' = \text{Reduce}(C, \text{red})$ for some reduction tuple red . Let $\text{Cost}(\text{Reduce}(C))$ and $\text{Cost}(\text{PP}(C, C'))$ denote the costs of $\text{Reduce}(C, \text{red})$ and $\text{PairProduct}(C, C')$, respectively. Then $\text{Cost}(\text{Reduce}(C)) \leq \text{Cost}(\text{PP}(C, C'))$.*

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	$C' = \text{Reduce}(C, \text{red})$	$\text{PairProduct}(C, C')$
ForkGouping	[1, 1]	DontCareGrouping	[ForkGouping, ([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 $\text{Reduce}(C, \text{red})$, which justifies saying that $\text{Cost}(\text{Reduce}(C)) \leq \text{Cost}(\text{PP}(C, C'))$.

Induction step:

Induction Hypothesis: Assume that for all level- k proto-CFLOBDDs C'_k and C_k for which $C'_k = \text{Reduce}(C_k, \text{red})$, for some reduction tuple red , $\text{Cost}(\text{Reduce}(C_k)) \leq \text{Cost}(\text{PP}(C_k, C'_k))$.

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

(i) A-connections. PairProduct is first called recursively on $C_{k+1}.A$ and $C'_{k+1}.A$ —i.e., the level- k A-connections of C_{k+1} and C'_{k+1} , respectively. By the construction of C'_{k+1} from C_{k+1} , we know that $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, \text{red}_A)$ for some reduction tuple red_A . Thus, by the induction hypothesis,

$$\text{Cost}(\text{Reduce}(C_{k+1}.A)) \leq \text{Cost}(\text{PP}(C_{k+1}.A, C'_{k+1}.A)). \quad (42)$$

(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 $[\tilde{C}_{k+1}.A, \text{midVertexPairs}]$. By the construction of C'_{k+1} from C_{k+1} , we know that $C'_{k+1}.A = \text{Reduce}(C_{k+1}.A, \text{red}_A)$ for some reduction tuple red_A , and thus by Thm. L.1, $\tilde{C}_{k+1}.A = C_{k+1}.A$.

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

We now consider the meaning of the pairs $(i, j) \in \text{midVertexPairs}$ from the standpoint of the language partitions used in Thm. L.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, \dots, 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 $\{i_1, i_2, \dots, i_p\}$ 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 $\text{Reduce}(C_{k+1}, \text{red})$ can only be because there were calls on $\text{Reduce}(C_{k+1}.B[i_1], \text{red}_1)$, $\text{Reduce}(C_{k+1}.B[i_2], \text{red}_2), \dots, \text{Reduce}(C_{k+1}.B[i_p], \text{red}_p)$, for which the results were all equal to $C'_{k+1}.B[j]$. (The fifth bullet point of Ex. L.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,

$$\begin{aligned} \text{Cost}(\text{Reduce}(C_{k+1}.B[i_1])) &\leq \text{Cost}(\text{PP}(C_{k+1}.B[i_1], C'_{k+1}.B[j])) \\ \text{Cost}(\text{Reduce}(C_{k+1}.B[i_2])) &\leq \text{Cost}(\text{PP}(C_{k+1}.B[i_2], C'_{k+1}.B[j])) \\ &\dots \\ \text{Cost}(\text{Reduce}(C_{k+1}.B[i_p])) &\leq \text{Cost}(\text{PP}(C_{k+1}.B[i_p], C'_{k+1}.B[j])) \end{aligned} \tag{43}$$

Let e_A denote the number of exit vertices of $C_{k+1}.A$ (which is also the number of middle vertices of C_{k+1}). These inequalities can be expressed more succinctly by observing that for each index i , $1 \leq i \leq e_A$ on the left-hand side (corresponding to an A-connection language-partition of $C_{k+1}.A$), there is a unique j to use on the right-hand side of the inequality. (Index j corresponds to the coarsened A-connection language-partition of $C'_{k+1}.A$.) Let *reductum* denote this index map: i.e., $j = \text{reductum}(i)$. We can now rewrite Eqn. (43) as

$$\text{Cost}(\text{Reduce}(C_{k+1}.B[i])) \leq \text{Cost}(\text{PP}(C_{k+1}.B[i], C'_{k+1}.B[\text{reductum}(i)])) \tag{44}$$

- (iii) Overall cost.** Let g' and g denote the outermost groupings (at level $k+1$) of C'_{k+1} and C_{k+1} , respectively. Reduce and PairProduct each make a call on *RepresentativeGrouping* at the end of their computations to hash-cons the outermost grouping that has been constructed. The time complexity of a call on *RepresentativeGrouping* is dominated by the cost of computing the grouping's hash value, and thus the costs in Reduce and PairProduct are linear in $|g'|$ and $|g|$, respectively. By Lem. L.2, we know that $|g'| \leq |g|$, and thus the cost of the call on *RepresentativeGrouping* in Reduce is no more than the cost of the call in PairProduct.

Finally, using Lem. L.2 and Eqns. (42) and (44), we obtain the desired result:

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

□