Chain Reduction for Binary and Zero-Suppressed Decision Diagrams

Randal E. Bryant[(✉)]

Computer Science Department, Carnegie Mellon University,
Pittsburgh, PA 15213, USA
Randy.Bryant@cs.cmu.edu

Abstract. Chain reduction enables reduced ordered binary decision diagrams (BDDs) and zero-suppressed binary decision diagrams (ZDDs) to each take advantage of the others' ability to symbolically represent Boolean functions in compact form. For any Boolean function, its chain-reduced ZDD (CZDD) representation will be no larger than its ZDD representation, and at most twice the size of its BDD representation. The chain-reduced BDD (CBDD) of a function will be no larger than its BDD representation, and at most three times the size of its CZDD representation. Extensions to the standard algorithms for operating on BDDs and ZDDs enable them to operate on the chain-reduced versions. Experimental evaluations on representative benchmarks for encoding word lists, solving combinatorial problems, and operating on digital circuits indicate that chain reduction can provide significant benefits in terms of both memory and execution time.

1 Introduction

Decision diagrams (DDs) encode sets of values in compact forms, such that operations on the sets can be performed on the encoded representation, without expanding the sets into their individual elements. In this paper, we consider two classes of decision diagrams: reduced ordered binary decision diagrams (BDDs) [4] and zero-suppressed binary decision diagrams (ZDDs) [11,12]. These two representations are closely related to each other, with each achieving more compact representations for different classes of applications. We present extensions to both representations, such that BDDs can take advantage of the source of compaction provided by ZDDs, and vice-versa.

Both BDDs and ZDDs encode sets of binary sequences of some fixed length n, defining a Boolean function over n variables. We can bound their relative sizes as follows. Suppose for some function, we encode it according to the different DD types. For function f, let $T(f)$ indicate the number of nodes (including leaf nodes) in the representation of type T. Let $R_f(T_1, T_2)$ denote the relative sizes when representing f using types T_1 and T_2:
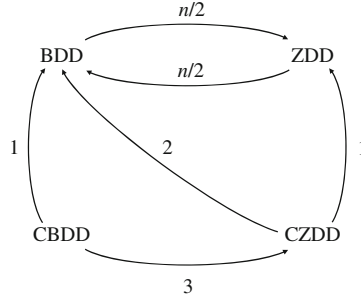
$$R_f(T_1, T_2) = \frac{T_1(f)}{T_2(f)}$$

Fig. 1. Size bound relations between different representations

Comparing BDDs and ZDDs, Knuth [9] has shown that for any function f:

$$R_f(\mathrm{BDD}, \mathrm{ZDD}) \leq n/2 + o(n) \tag{1}$$

$$R_f(\mathrm{ZDD}, \mathrm{BDD}) \leq n/2 + o(n) \tag{2}$$

As these bounds show, ZDDs may be significantly (a factor of $n/2$) more compact than BDDs, or vice-versa. In practice, the comparative advantage of one representation over the other can be very significant, given that the size of the data structure is often the limiting factor in the use of DDs.

In this paper, we introduce two new representations: *chain-reduced ordered binary decision diagrams* (CBDDs), and *chain-reduced zero-suppressed binary decision diagrams* (CZDDs). The key idea is to associate two levels with each node and to use such nodes to encode particular classes of linear chains found in BDDs and ZDDs. Chain reduction can be defined in terms of a set of reduction rules applied to BDDs and ZDDs, giving bounds for any function f

$$R_f(\mathrm{CBDD}, \mathrm{BDD}) \leq 1 \tag{3}$$

$$R_f(\mathrm{CZDD}, \mathrm{ZDD}) \leq 1 \tag{4}$$

We show bounds on the relative sizes of the representations as:

$$R_f(\mathrm{CBDD}, \mathrm{CZDD}) \leq 3 \tag{5}$$

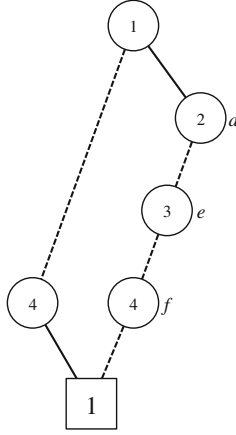$$R_f(\mathrm{CZDD}, \mathrm{BDD}) \leq 2 \tag{6}$$

These relations are summarized in the diagram of Fig. 1. In this figure, each arc from type T_1 to type T_2 labeled by an expression E indicates that $R_f(T_1, T_2) \leq E + o(E)$. We also show these bounds are tight, by demonstrating parameterized families of functions that achieve the bounding factors of (5) and (6).

These results indicate that the two compressed representations will always be within a small constant factor (2 for CZDDs and 3 for CBDDs) of either a BDD or a ZDD representation. While one representation may be more slightly compact than the other, the relative advantage is bounded by a constant factor, and hence choosing between them is less critical.
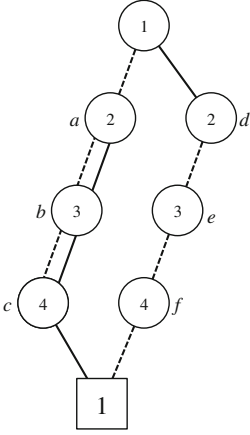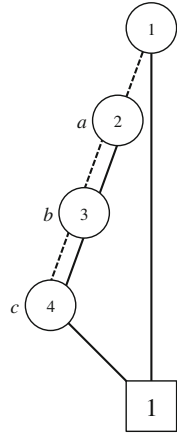
(A) Levelized BDD   (B) BDD   (C) ZDD

Fig. 2. Reductions in BDDs and ZDDs. Each reduces the representation size with edges between nonconsecutive levels.

This paper defines the two compressed representations, derives the bounds indicated in (5) and (6) and presents extensions of the core BDD and ZDD algorithms to their chained versions. It describes an implementation based on modifications of the CUDD BDD package [14]. It presents some experimental results and concludes with a discussion of the merits of chaining and possible extensions.

2   Related Work

In independent work, van Dijk and his colleages devised a hybrid of BDDs and ZDDs they call *tagged BDDs* [6]. Their representation augments BDDs by associating a variable with each edge, in addition to the variable associated with each node, enabling them to represent both BDD and ZDD reductions along each edge. For any function, a tagged BDD is guaranteed to have no more nodes than either its BDD or its ZDD representation. They avoid the constant factor in node growth that CBDDs or CZDDs may require, at the cost of requiring storage for three variables per node (one for the node, and one for each of the outgoing edges) versus two. Choosing between their representation or ours depends on a number of implementation factors. Both achieve the larger goal of exploiting the reductions enabled by both BDDs and ZDDs.

3   BDDs and ZDDs

Both BDDs and ZDDs encode sets of binary sequences of length n as directed acyclic graphs with two leaf nodes, labeled with values **0** and **1**, which we refer

to as "leaf **0**" and "leaf **1**," respectively. Each nonleaf node v has an associated level l, such that $1 \leq l \leq n$, and two outgoing edges, labeled *lo* and *hi* to either a leaf node or a nonleaf node. By convention, leaf nodes have level $n + 1$. An edge from v to node u having level l' must have $l < l'$.

Figure 2 shows three decision-diagram representations of the set S, defined as:

$$S = \{0001, 0011, 0101, 0111, 1000\} \tag{7}$$

The *lo* edge from each node is shown as a dashed line, and the *hi* edge is shown as a solid line. As a shorthand, we omit leaf **0** and all branches to it.

Graph A represents S as a *levelized binary decision diagram*, where an edge from a node with level l must connect to either leaf **0** or to a node with level $l + 1$. Each path from the root to leaf **1** encodes an element of set S. For a given path, the represented sequence has value 0 at position l when the path follows the *lo* edge from the node with level l and value 1 when the path follows the *hi* edge.

Graph A has nodes forming two linear chains: a DON'T-CARE *chain*, consisting of nodes a and b, and an OR *chain*, consisting of nodes d, e, and f. A DON'T-CARE chain is a series of DON'T-CARE nodes, each having its two outgoing edges directed to the same next node. In terms of the set of represented binary sequences, a DON'T-CARE node with level l allows both values 0 and 1 at sequence position l. An OR chain consists of a sequence where the outgoing *hi* edges for the nodes all go the same node—in this case, leaf **0**. An OR chain where all *hi* edges lead to leaf **0** has only a single path, assigning value 0 to the corresponding positions in the represented sequence. We will refer to this special class of OR chain as a ZERO *chain*.

BDDs and ZDDs differ from each other in the interpretations they assign to a *level-skipping edge*, when a node with level l has an edge to a node with level l' such that $l + 1 < l'$. For BDDs, such an edge is considered to encode a DON'T-CARE chain. Thus, graph B in Fig. 2 shows an BDD encoding set S. The edge on the left from level 1 to level 4 is equivalent to the DON'T-CARE chain formed by nodes a and b of graph A. For ZDDs, a level skipping edge encodes a ZERO chain. Thus, graph C shows a ZDD encoding set S. The edge on the right from level 1 to the leaf encodes the ZERO chain formed by nodes d, e, and f of graph A. Whether the set is encoded as a BDD or a ZDD, one type of linear chains remains. Introducing chain reduction enables BDDs and ZDDs to exploit both DON'T-CARE and OR (and therefore ZERO) chains to compress their representations.

4 Chain Patterns and Reductions

Figure 3 shows the general form of OR and DON'T-CARE chains, as were illustrated in the examples of Fig. 2. These chains have levels ranging from t to b, such that $1 \leq t < b \leq n$. Each form consists of a linear chain of nodes followed by two nodes f and g with levels greater than b. Nodes f and g are drawn as

triangles to indicate that they are the roots of two subgraphs in the representation. In an OR chain, the *lo* edge from each node is directed to the next node in the chain, and the *hi* edge is directed to node *g*. The chains eliminated by ZDDs are a special case where $g = \mathbf{0}$. In a DON'T-CARE chain, both the *lo* and the *hi* edges are directed toward the next node in the chain.

As was illustrated in Fig. 2, having edges that skip levels allows BDDs to compactly represent DON'T-CARE chains and ZDDs to eliminate OR chains when $g = \mathbf{0}$. The goal of chain reduction is to allow both forms to compactly represent both types of chains. They do so by associating two levels with each node, as indicated in Fig. 3(C). That is, every nonleaf node has an associated pair of levels $t\!:\!b$, such that $1 \leq t \leq b \leq n$. In a *chain-reduced ordered binary decision diagram* (CBDD), such a node encodes the OR chain pattern shown in Fig. 3(A), while in a *chain-reduced zero-suppressed binary decision diagram* (CZDD), such a node encodes the DON'T-CARE chain pattern shown in Fig. 3(B). A node with levels t and b such that $t = b$ encodes a standard node with respect to the indicated variable.
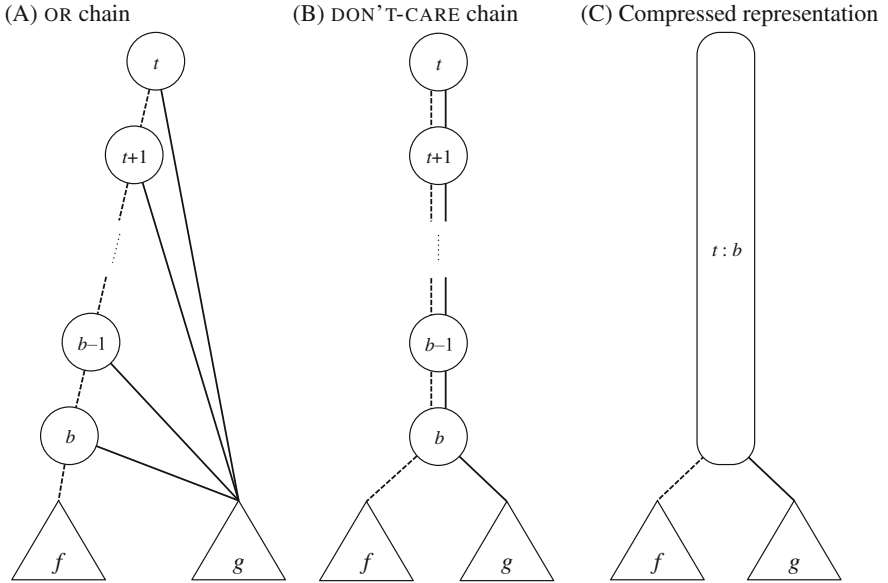

Fig. 3. Chain patterns. These patterns remain after BDD reduction (A), and ZDD reduction (B), but can be represented in compressed form (C).

Figure 4 shows the effect of chain reduction, starting with the levelized graph A. In the CBDD (B), a single node f' replaces the OR chain consisting of nodes d, e, and f. In the CZDD (C), the DON'T-CARE chain consisting of nodes a and b is incorporated into node c to form node c'. These new nodes are drawn in elongated form to emphasize that they span a range of levels, but it should be emphasized that *all* nodes in a chained representation have an associated pair of levels.
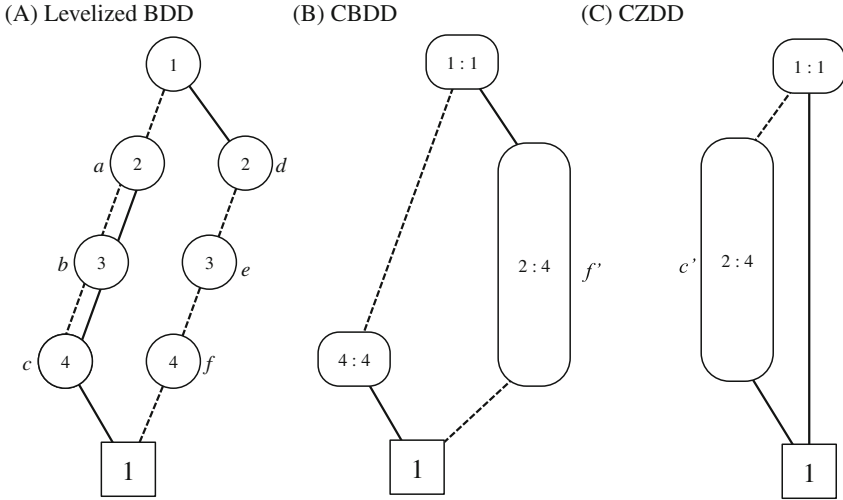
Fig. 4. Chain Reduction Examples. Each now reduces both chain types.

To generalize from these examples, let us denote a node of the form illustrated in Fig. 3(C) with the modified if-then-else notation $\langle t : b \to g, f \rangle$. That is, the node has a range of levels from t to b, an outgoing *hi* edge to node g, and an outgoing *lo* edge to node f.

A BDD representation of a function can be transformed into a CBDD as follows. The process starts by labeling each node having level l in the BDD with the pair $t : b$, such that $t = b = l$. Then, we repeatedly apply a *reduction rule*, replacing any pair of nodes u and v of the form $u = \langle t : m \to g, v \rangle$ and $v = \langle m + 1 : b \to g, f \rangle$ by the single node $\langle t : b \to g, f \rangle$. A similar process can transform any ZDD representation of a function into a CZDD, using the reduction rule that a pair of nodes u and v of the form $u = \langle t : m \to v, v \rangle$ and $v = \langle m + 1 : b \to g, f \rangle$ is replaced by the single node $\langle t : b \to g, f \rangle$. In practice, most algorithms for constructing decision diagrams operate from the bottom up. The reduction rules are applied as nodes are created, and so unreduced nodes are never actually generated.

5   Size Ratio Bounds

These reduction rules allows us to bound the relative sizes of the different representations, as given by (5) and (6).

First, let us consider (5), bounding the relative sizes of the CBDD and CZDD representations of a function. Consider a graph G representing function f as a CZDD. We can generate a CBDD representation G' as follows. G' contains a node v' for each node v in G. However, if v has levels $t : b$, then v' has levels $b : b$, because any DON'T-CARE chain encoded explicitly in the CZDD is encoded implicitly in a CBDD.

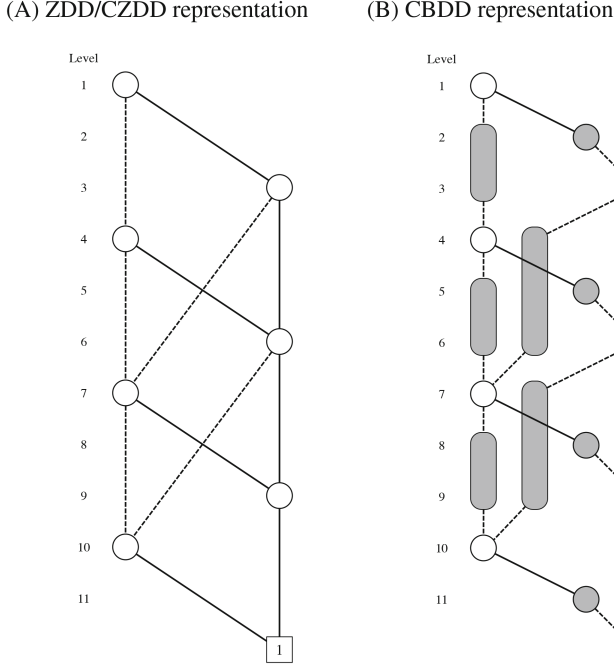(A) ZDD/CZDD representation   (B) CBDD representation

Fig. 5. Worst case example for effectiveness of CBDD compression. The implicit ZERO chains in the ZDD (A) must be explicitly encoded in the CBDD (B), increasing its size by a factor of 3.

Consider an edge from node u to node v in G, where the nodes have levels $t_u : b_u$ and $t_v : b_v$, respectively. If $t_v = b_u + 1$, then there can be an edge directly from u' to v'. If $t_v < b_u + 1$, then we introduce a new node to encode the implicit zero chain in G from u to v. This node has the form $\langle b_u + 1 : t_v - 1 \rightarrow \mathbf{0}, v' \rangle$ and has an edge from u' to it.

The size of G' is bounded by the number of nodes plus the number of edges in G. Since each node in G has at most two outgoing edges, we can see that G' has at most three times the number of nodes as G. Graph G' may not be reduced, but it provides an upper bound on the size of a CBDD relative to that of a CZDD.

This bound is tight—Fig. 5 illustrates the reduced representations for a family of functions, parameterized by a value k ($k = 3$ in the example), such that the function is defined over $3k + 2$ variables. The ZDD and CZDD representations are identical (A), having $2k + 3$ nodes (including both leaf nodes.) The CBDD representation has $6k + 2$ nodes (B). We can see in this example that the CBDD requires nodes (shown in gray) to encode the ZERO chains that are implicit in the ZDD.

Second, let us consider (6), bounding the relative sizes of the CZDD and BDD representations of a function. Consider a graph G representing function f

as a BDD. We can construct its representation G' as a CZDD. Consider each edge G from node u, having level l_u to node v, having level l_v. Let $r = lo(v)$ and $s = hi(v)$. G' has a node w_{uv} of the form $\langle l_u + 1 : l_v \rightarrow w_{vs}, w_{vr} \rangle$. That is, w_{uv} encodes any DON'T-CARE chain between u and v, and it has edges to the nodes generated to encode the edges between v and its two children. The size of G' is bounded by the number of edges in G, which is at most twice the number of nodes.

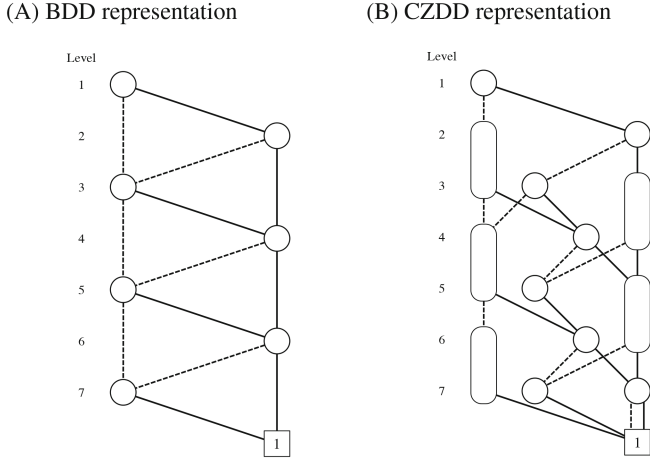(A) BDD representation   (B) CZDD representation

Fig. 6. Worst case example for effectiveness of CZDD compression. The nodes in the BDD (A) must be duplicated to encode the incoming DON'T-CARE chains (B), increasing the size by a factor of 2.

This bound is also tight—Fig. 6 illustrates the reduced representations for a family of functions, parameterized by a value k ($k = 3$ in the example), such that the function is defined over $2k + 1$ variables. The BDD representations (A) has $2k + 3$ nodes (including both leaf nodes). The CZDD representation has $4k + 3$ nodes (B). We can see that most of the nodes in the BDD must be duplicated: once with no incoming DON'T-CARE chain, and once with a chain of length one.

As can be seen in Fig. 1, these bounds contain an asymmetry between BDDs and ZDDs and their compressed forms. The bound of 3 holds between CBDDs and CZDDs, and hence by transitivity between CBDDs and ZDDs, while the bound of 2 holds only between CZDDs and BDDs. The general form of the OR chain (Fig. 3(A)), where g is something other than **0**, cannot be directly encoded with CZDD nodes.

6   Operating on CBDDs and CZDDs

The APPLY algorithms for decision diagrams operate by recursively expanding a set of argument decision diagrams according to a Shannon expansion of the

represented functions [4,5]. These algorithms allow functions to be combined according to standard binary Boolean operations, as well as by the if-then-else operation ITE.

As notation, consider a step that expands k argument nodes $\{v_i | 1 \leq i \leq k\}$ where $v_i = \langle t_i : b_i \rightarrow g_i, f_i \rangle$. For example, operations AND, OR, and XOR use the APPLY algorithm with $k = 2$, while ternary operations, such as ITE use $k = 3$. A step can be summarized as follows:

1. If one of the terminal cases apply, then return the result.
2. If the computed cache contains an entry for this combination of operation and arguments, then return the previously computed result.
3. Recursively compute the result:
   (a) Choose splitting level(s) based on the levels of the arguments.
   (b) Generate hi and lo cofactors for each argument.
   (c) Recursively compute the hi and lo values of the result using the APPLY algorithm with the hi cofactors and the lo cofactors, respectively.
   (d) Determine the result node parameters based on the computed hi and lo cofactors, the splitting level(s), and the reduction rules.
   (e) Either reuse an existing node or create a new one with the desired level(s) and hi and lo children.
4. Store an entry in the computed cache.
5. Return the computed value.

In generalizing from conventional BDDs and ZDDs to their chained versions, we need only modify 3(a) (splitting), 3(b) (cofactoring), and 3(d) (combining) in this sequence. In the following presentation, we first give formal definitions and then provide brief explanations.

For CBDDs, we define the splitting levels t and b as:

$$t = \min_{1 \leq i \leq k} t_i \qquad (8)$$

$$b = \min_{1 \leq i \leq k} \begin{cases} b_i, & t_i = t \\ t_i, & t_i = n + 1 \\ t_i - 1, & \text{else} \end{cases}$$

We then define the two cofactors for each argument node v_i, denoted $lo(v_i, t : b)$ and $hi(v_i, t : b)$, according to the following table:

Case	Condition	$lo(v_i, t : b)$	$hi(v_i, t : b)$
1	$b < t_i$	v_i	v_i
2	$b = b_i$	f_i	g_i
3	$t_i \leq b < b_i$	$\langle b + 1 : b_i \rightarrow g_i, f_i \rangle$	g_i

These three cases can be explained as follows:

Case 1: Splitting spans levels less than the top level of v_i. Since level-skipping edges encode DON'T-CARE chains, both cofactors equal the original node.

Case 2: Splitting spans the same levels as node v_i. The cofactors are therefore the nodes given by the outgoing edges.

Case 3: Splitting spans a subset of the levels covered by node v_i. We construct a new node spanning the remaining part of the encoded OR chain for the *lo* cofactor and have g_i as the *hi* cofactor.

Recursive application of the APPLY operation on the cofactors generates a pair of nodes u_0 and u_1. Using the variable levels t and b defined in (8), these are combined to form a result node u, defined as follows:

$$u = \begin{cases} u_0, & u_0 = u_1 & \text{Case 1} \\ \langle t : b' \rightarrow u_1, w_0 \rangle, & u_0 = \langle b+1, b' : \rightarrow u_1, w_0 \rangle & \text{Case 2} \\ \langle t : b \rightarrow u_1, u_0 \rangle, & \text{else} & \text{Case 3} \end{cases} \tag{9}$$

These three cases can be explained as follows:

Case 1: The *hi* and *lo* cofactors are identical, and so the don't-care reduction rule can be applied.

Case 2: Chain compression can be applied to create a node that absorbs the *lo* cofactor.

Case 3: No special rules apply.

Similar rules hold for applying operations to CZDDs, although there are important differences, due to the different interpretations of level-skipping edges.

We define the splitting levels t and b as:

$$t = \min_{1 \leq i \leq k} t_i \tag{10}$$

$$b = \min_{1 \leq i \leq k} \begin{cases} b_i, & t_i = t \\ n+1, & v_i = \mathbf{0} \\ t, & \text{else} \end{cases}$$

The cofactors for argument node v_i are defined according to the following table:

Case	Condition	$lo(v_i, t : b)$	$hi(v_i, t : b)$
1	$b < t_i$	v_i	$\mathbf{0}$
2	$b = b_i$	f_i	g_i
3	$t_i \leq b < b_i$	$\langle b+1 : b_i \rightarrow g_i, f_i \rangle$	$\langle b+1 : b_i \rightarrow g_i, f_i \rangle$

These three cases can be explained as follows:

Case 1: The splitting spans levels less than the top level of v_i. Since level-skipping edges encode ZERO chains, the *lo* cofactor equals the original node and the *hi* cofactor equals leaf **0**.

Case 2: The splitting spans the same levels as node v_i. The cofactors are therefore the nodes given by the outgoing edges.

Case 3: The splitting spans a subset of the levels covered by node v_i. We construct a new node spanning the remaining part of the encoded DON'T-CARE chain for both cofactors.

Recursive application of the APPLY operation on the cofactors generates a pair of nodes u_0 and u_1. Using the variable ranges t and b defined in (10), these are combined to form a result node u, defined as follows:

$$
u = \begin{cases}
u_0, & u_1 = \mathbf{0} \text{ and } t = b & \text{Case 1} \\
\langle t : b - 1 \to u_0, u_0 \rangle, & u_1 = \mathbf{0} \text{ and } t < b & \text{Case 2} \\
\langle t : b' \to w_1, w_0 \rangle, & u_0 = u_1 = \langle b + 1, b' : \to w_1, w_0 \rangle & \text{Case 3} \\
\langle t : b \to u_1, u_0 \rangle, & \text{else} & \text{Case 4}
\end{cases}
\tag{11}
$$

These four cases can be explained as follows:

Case 1: The zero-suppression rule can be applied to return a direct pointer to u_0

Case 2: The zero-suppression rule can be applied, but we must construct a node encoding the DON'T-CARE chain between levels t and $b - 1$.

Case 3: Chain compression can be applied to create a node that absorbs the *lo* cofactor.

Case 4: No special rule applies.

7   Experimental Results

We implemented both CBDDs and CZDDs by modifying version 3.0.0 of the CUDD BDD package [14]. When compiled for 64-bit execution, CUDD stores a 32-bit field `index` in each node, where this index is translated into a level according to the variable ordering. For our implementation, we split this field into two 16-bit fields `index` and `bindex` to (indirectly) encode the top and bottom levels of the node. Thus, there was no storage penalty for the generalization to a chained form.

CUDD uses complement edges when representing BDDs [2,13]. Complement edges can potentially reduce the size of a BDD by a factor of two, invalidating the size ratio bounds derived in (5) and (6). For our experimental results, we therefore use a representation based on CUDD's support for *Algebraic Decision Diagrams* (ADDs) [1]. ADDs generalize BDDs by allowing arbitrary leaf values. Restricting the leaf values to 0 and 1 yields conventional BDDs without complement edges.

To evaluate the effectiveness of chain reduction, we chose three different categories of benchmarks to compare the performance of BDDs, ZDDs, and their chained versions. One set of benchmarks evaluated the ability of DDs to represent information in compact form, a second to evaluate their ability to solve combinatorial problems, and a third to represent typical digital logic functions. All experiments were performed on a 4.2 GHz Intel Core i7 processor with 32 GB of memory running the OS X operating system.

7.1   Encoding a Dictionary

As has been observed [9], a list of words can be encoded as a function mapping strings in some alphabet to either 1 (included in list) or 0 (not included in list). Strings can further be encoded as binary sequences by encoding each symbol as a sequence of bits, allowing the list to be represented as a Boolean function. We consider two possible encodings of the symbols, defining the *radix r* to be the number of possible symbols. A *one-hot* encoding (also referred to as a "1-of-N" encoding) requires r bits per symbol. Each symbol is assigned a unique position, and the symbol is represented with a one in this position and zeros in the rest. A *binary* encoding requires $\lceil \log_2 r \rceil$ bits per symbol. Each symbol is assigned a unique binary pattern, and the symbol is represented by this pattern. Lists consisting of words with multiple lengths can be encoded by introducing a special "null" symbol to terminate each word.

Eight benchmarks were derived from two word lists to allow comparisons of different encoding techniques and representations. The first list is a set of English words in the file `/usr/share/dict/words` found on Macintosh systems. It contains 235,886 words with lengths ranging from one to 24 symbols, and where the symbols consist of lower- and upper-case letters plus hyphen. We consider two resulting symbol sets: a *compact* form, consisting of just the symbols found in the words plus a null symbol (54 total), and an *ASCII* form, consisting of all 128 ASCII characters plus a null symbol. The second word list is from an online list of words employed by password crackers. It consists of 979,247 words ranging in length from one to 32 symbols, and where the symbols include 79 possible characters. Again, we consider both a compact form and an ASCII form. The choice of one-hot vs. binary encoding has a major effect on the number of Boolean variables required to encode the words. With a one-hot encoding, the number of variables ranges between 1,296 and 4,128, while it ranges between 144 and 256 with a binary representation. To generate DD encodings of a word list, we first constructed a trie representation the words and then generated Boolean formulas via a depth-first traversal of the trie.

Figure 7 shows the number of nodes required to represent word lists as Boolean functions, according to the different lists, encodings, and DD types. The entry labeled "(C)ZDD" gives the node counts for both ZDDs and CZDDs. These are identical, because there were no DON'T-CARE chains for these functions. The two columns on the right show the ratios between the different DD types. Concentrating first on one-hot encodings, we see that the chain compression of CBDDs reduces the size compared to BDDs by large factors (15.50–34.03).

One-hot	Node counts			Ratios	
	BDD	CBDD	(C)ZDD	BDD:CBDD	CBDD:CZDD
Compact word list	9,701,439	626,070	297,681	15.50	2.10
ASCII word list	23,161,501	626,071	297,681	37.00	2.10
Compact password list	49,231,085	2,321,572	1,130,729	21.21	2.05
ASCII password list	79,014,931	2,321,792	1,130,729	34.03	2.05

Binary	Node counts			Ratios	
	BDD	CBDD	(C)ZDD	BDD:CBDD	CBDD:CZDD
Compact word list	1,117,454	1,007,868	723,542	1.11	1.39
ASCII word list	1,464,773	1,277,640	851,580	1.15	1.50
Compact password list	4,422,292	3,597,474	2,506,088	1.23	1.44
ASCII password list	4,943,940	4,307,614	2,875,612	1.15	1.50

Fig. 7. Node counts and ratios of node counts for dictionary benchmarks

Compared to ZDDs, representing the lists by CBDDs incurs some penalty (2.05–2.10), but less than the worst-case penalty of 3. Increasing the radix from a compact form to the full ASCII character set causes a significant increase in BDD size, but this effect is eliminated by using the zero suppression capabilities of CBDDs, ZDDs, and CZDDs.

Using a binary encoding of the symbols reduces the variances between the different encodings and DD types. CBDDs provide only a small benefit (1.11–1.23) over BDDs, and CBDDs are within a factor of 1.50 of ZDDs. Again, chaining of ZDDs provides no benefit. Observe that for both lists, the most efficient representation is to use either ZDDs or CZDDs with a one-hot encoding. The next best is to use CBDDs with a one-hot encoding, and all three of these are insensitive to changes in radix. These cases demonstrate the ability of ZDDs (and CZDDs) to use very large, sparse encodings of values. By using chaining, CBDDs can also take advantage of this property.

Although the final node counts for the benchmarks indicate no advantage of chaining for ZDDs, statistics characterizing the effort required to derive the functions show a significant benefit. Figure 8 indicates the total number of operations and the total time required for generating ZDD and CZDD representations of the benchmarks. The operations are computed as the number of times the program checks for an entry in the operation cache (step 2 in the description of the APPLY algorithm). There are many operational factors that can affect the number of operations, including the program's policies for operation caching and garbage collection. Nevertheless, it is some indication of the amount of activity required to generate the DDs. We can see that chaining reduces the number of operations by factors of 8.87–13.30. The time required depends on many attributes of the DD package and the system hardware and software. Here we see that chaining improves the execution time by factors of 1.35–15.26.

With unchained ZDDs, many of the intermediate functions have large DON'T-CARE chains. For example, the ZDD representation of the function x, for variable x, requires $n + 2$ nodes—one for the variable, $n - 1$ for the DON'T-CARE chains

One-hot	Operations			Time (secs.)		
	ZDD	CZDD	Ratio	ZDD	CZDD	Ratio
Compact word list	142,227,877	12,097,435	11.76	48.78	15.04	3.24
ASCII word list	375,195,184	28,574,814	13.13	173.56	21.84	7.95
Compact password list	806,017,001	62,785,274	12.84	713.15	46.73	15.26
ASCII password list	1,383,534,557	104,059,626	13.30	658.21	57.81	11.39

Binary	Operations			Time (secs.)		
	ZDD	CZDD	Ratio	ZDD	CZDD	Ratio
Compact word list	15,701,738	1,826,171	8.60	13.11	9.70	1.35
ASCII word list	20,921,746	2,139,574	9.78	14.40	10.20	1.41
Compact password list	66,489,058	7,499,615	8.87	52.52	30.62	1.72
ASCII password list	75,556,080	7,936,321	9.52	50.77	30.33	1.67

Fig. 8. Impact of chaining on effort required to generate DD representations of word lists.

before and after the variable, and two leaf nodes. With chaining, this function reduces to just four nodes: the upper DON'T-CARE chain is incorporated into the node for the variable, and a second node encodes the lower chain. Our dictionary benchmarks have over 4,000 variables, and so some of the intermediate DDs can be more than 1,000 times more compact due to chaining.

7.2   The 15-Queens Problem

A second set of benchmarks involved representing all possible solutions to the n-queens problem [12] as a Boolean function. This problem attempts to place n queens on a $n \times n$ chessboard in such a way that no two queens can attack each other. For our benchmark we chose $n = 15$ to stay within the memory limit of the processor being used.

Once again, there are two choices for encoding the positions of queens on the board. A *one-hot* encoding uses a Boolean variable for each square. A *binary* encoding uses $\lceil \log_2 n \rceil = 4$ variables for each row, encoding the position of the queen within the row.

Our most successful approach for encoding the constraints with Boolean operations worked from the bottom row to the top. At each level, it generated formulas for each column and diagonal expressing whether it was occupied in the rows at or below this one, based on the formulas for the level below and the variables for the present row.

We considered two ways of ordering the variables for the different rows. The *top-down* ordering listed the variables according to the row numbers 1 through 15. The *center-first* ordering listed variables according to the following row number sequence:

$$8, 9, 7, 10, 6, 11, 5, 12, 4, 13, 3, 14, 2, 15, 1.$$

Our hope was that ordering the center rows first would reduce the DD representation size. This proved not to be the case, but the resulting node counts are instructive.

One-hot		Node counts			Ratios	
Ordering	Graph(s)	BDD	CBDD	CZDD	BDD:CBDD	CBDD:CZDD
Top-down	Final	51,889,029	10,529,738	4,796,504	4.93	2.20
Top-down	Peak	165,977,497	39,591,936	18,625,659	4.19	2.13
Center-first	Final	65,104,658	12,628,086	5,749,613	5.16	2.20
Center-first	Peak	175,907,712	42,045,602	19,434,105	4.18	2.16

Binary		Node counts			Ratios	
Ordering	Graph(s)	BDD	CBDD	CZDD	BDD:CBDD	CBDD:CZDD
Top-down	Final	13,683,076	11,431,403	7,383,739	1.20	1.55
Top-down	Peak	43,954,472	38,898,146	26,682,980	1.13	1.46
Center-first	Final	17,121,947	14,185,276	9,054,115	1.21	1.57
Center-first	Peak	46,618,943	41,362,659	28,195,596	1.13	1.47

Fig. 9. Node counts and ratios of node counts for 15-queens benchmarks

Figure 9 shows the node counts for the different benchmarks. It shows both the size of the final function representing all solutions to the 15-queens problem, as well as the peak size, computed as the maximum across all rows of the combined size of the functions that are maintained to express the constraints imposed by the row and those below it. For both the top-down and the center-first benchmarks, this maximum was reached after completing row 3. Typically the peak size was around three times larger than the final size.

For a one-hot encoding, we can see that CBDDs achieve factors of 4.18–5.16 compaction over BDDs, and they come within a factor of 2.20 of CZDDs. For a binary encoding, the levels of compaction are much less compelling (1.13–1.20), as is the advantage of CZDDs over BDDs. It is worth noting that the combination of a one-hot encoding and chaining gives lower peak and final sizes than BDDs with a binary encoding.

One-hot		Node counts		Ratios
Ordering	Graph(s)	ZDD	CZDD	ZDD:CZDD
Top-down	Final	4,796,504	4,796,504	1.00
Top-down	Peak	18,632,019	18,625,659	1.00
Center-first	Final	5,749,613	5,749,613	1.00
Center-first	Peak	73,975,637	19,434,105	3.81

Binary		Node counts		Ratios
Ordering	Graph(s)	ZDD	CZDD	ZDD:CZDD
Top-down	Final	7,383,739	7,383,739	1.00
Top-down	Peak	26,684,315	26,682,980	1.00
Center-first	Final	9,054,115	9,054,115	1.00
Center-first	Peak	33,739,362	28,195,596	1.20

Fig. 10. Effect of chaining for ZDD representations of 15-queens benchmarks

	Node counts			Ratios	
Circuit	BDD	ZDD	CZDD	ZDD:BDD	CZDD:BDD
c432	31,321	48,224	41,637	1.54	1.33
c499	49,061	49,983	48,878	1.02	1.00
c880	23,221	52,436	32,075	2.26	1.38
c1908	17,391	18,292	17,017	1.05	0.98
c2670	67,832	261,736	85,900	3.86	1.27
c3540	3,345,341	4,181,847	3,538,982	1.25	1.06
c5315	636,305	898,912	681,440	1.41	1.07
c6288	48,181,908	48,331,495	48,329,117	1.00	1.00
c7552	4,537	37,689	4,774	8.31	1.05

Fig. 11. Node counts and ratios of node counts for digital circuit benchmarks

Figure 10 compares the sizes of the ZDD and CZDD representations of the 15-queens functions. We can see that the final sizes are identical—there are no DON'T-CARE chains in the functions encoding problem solutions. For the top-down ordering, CZDDs also offer only a small advantage for the peak requirement. For the center-first ordering, especially with a one-hot encoding, however, we can see that CZDDs are significantly ($3.81\times$) more compact. As the construction for row 3 completes, the variables that will encode the constraints for rows 2 and 5 remain unconstrained, yielding many DON'T-CARE chains. Once again, this phenomenon is much smaller with a binary encoding.

7.3   Digital Circuits

BDDs are commonly used in digital circuit design automation, for such tasks as verification, test generation, and circuit synthesis. We selected the circuits in the ISCAS '85 benchmark suite [3]. These were originally developed as benchmarks for test generation, but they have also been widely used as benchmarks for BDDs [7,10]. We generated variable orderings for all but last two benchmarks by traversing the circuit graphs, using the fanin heuristic of [10]. Circuit c6288 implements a 16×16 multiplier. For this circuit, the ordering of inputs listed in the file provided a feasible variable ordering, while the one generated by traversing the circuit exceeded the memory limits of our machine. For c7552, neither the ordering in the file, nor that provided by traversing the graph, generated a feasible order. Instead, we manually generated an ordering based on our analysis of a reverse-engineered version of the circuit described in [8]

Figure 11 presents data on the sizes of the DDs to represent all of the circuit outputs. We do not present any data for CBDDs, since these were all close in size to BDDs. We can see that the ZDD representations for these circuits are always larger than the BDD representations, by factors ranging up to 8.31. Using CZDDs mitigates that effect, yielding a maximum size ratio of 1.38.

8   Observations

Our experiments, while not comprehensive, demonstrate that chaining can allow BDDs to make use of large, sparse encodings, one of the main strengths of ZDDs. They also indicate that CZDDs may be the best choice overall. CZDDs have all of the advantages of ZDDs, while avoiding the risk of intermediate functions growing excessively large due to DON'T-CARE chains. They are guaranteed to stay within a factor of $2\times$ of BDDs. Even for digital circuit functions, we found this bound to be conservative—all of the benchmarks stayed within a factor of $1.4\times$.

References

1. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of the International Conference on Computer-Aided Design, pp. 188–191, November 1993
2. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proceedings of the 27th ACM/IEEE Design Automation Conference, pp. 40–45, June 1990
3. Brglez, F., Fujiwara, H.: A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In: 1985 International Symposium on Circuits And Systems (1985)
4. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
5. Bryant, R.E.: Binary decision diagrams. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-10575-8
6. van Dijk, T., Wille, R., Meolic, R.: Tagged BDDs: combining reduction rules from different decision diagram types. In: Formal Methods in Computer-Aided Design, pp. 108–115 (2017)
7. Fujita, M., Fujisawa, H., Kawato, N.: Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 2–5, November 1988
8. Hansen, M., Yalcin, H., Hayes, J.P.: Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering. IEEE Des. Test **16**(3), 72–80 (1999)
9. Knuth, D.E.: The Art of Computer Programming: Combinatorial Algorithms, Part I, vol. 4A. Addison Wesley, Reading (2011)
10. Malik, S., Wang, A., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Logic verification using binary decision diagrams in a logic synthesis environment. In: Proceedings of the International Conference on Computer-Aided Design, pp. 6–9, November 1988
11. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th ACM/IEEE Design Automation Conference, pp. 272–277, June 1993

12. Minato, S.: Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Academic Publishers, Norwell (1995)
13. Minato, S., Ishiura, N., Yajima, S.: Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In: Proceedings of the 27th ACM/IEEE Design Automation Conference, pp. 52–57, June 1990
14. Somenzi, F.: Efficient manipulation of decision diagrams. Int. J. Softw. Tools Technol. Transf. **3**(2), 171–181 (2001)