Noname manuscript No.
(will be inserted by the editor)

Chain Reduction for Binary and Zero-Suppressed Decision Diagrams

Randal E. Bryant

the date of receipt and acceptance should be inserted later

Abstract Chain reduction enables reduced ordered binary decision diagrams (BDDs) and zero-suppressed binary decision diagrams (ZDDs) to each take advantage of the other's 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. The experimental results are further validated by a quantitative model of how decision diagrams scale when encoding sets of sequences. This model explains why the combination of a one-hot encoding of the symbols in the sequences, plus a CBDD, CZDD, or ZDD representation of the set, yields the most compact form.

Keywords

Binary decision diagrams, zero-suppressed binary decision diagrams, Boolean functions

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) [5] and zero-suppressed binary decision diagrams (ZDDs) [21,22]. These two representations are closely related to each other, with each achieving more compact representations for different classes of applications. We

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

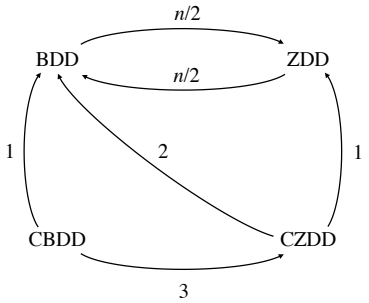


Fig. 1 Size bound relations between different representations

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:

$$R_f(T_1,T_2) = \frac{T_1(f)}{T_2(f)}$$

Comparing BDDs and ZDDs, Knuth [18] has shown that for any function f:

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

These bounds improve on the size ratios of n derived by Wegener [27]. 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 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. Like BDDs and ZDDs, both CBDDs and CZDDs provide canonical representations of Boolean functions. Chain reduction can be defined in terms of a set of reduction rules applied to BDDs and ZDDs, giving bounds for any function f

$$R_f(\mathrm{CBDD},\mathrm{BDD}) \leq 1 \quad (3)$$
$$R_f(\mathrm{CZDD},\mathrm{ZDD}) \leq 1 \quad (4)$$

In terms of graph sizes, using chain reduction can only lead to more compact representations.

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

$$R_f(\mathrm{CBDD},\mathrm{CZDD}) \leq 3 \quad (5)$$
$$R_f(\mathrm{CZDD},\mathrm{BDD}) \leq 2 \quad (6)$$

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 arcs define a transitive relation, and so we can also infer that $R_f(\mathrm{CBDD},\mathrm{ZDD}) \leq 3 + \mathrm{o}(1)$.

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 slightly more compact than the other, the relative advantage is bounded by a constant factor, and hence choosing between them is less critical than is the case when choosing between BDDs and ZDDs. The asymmetry in Fig 1—that CBDDs can compactly encode CZDDs, but not vice-versa—is explained in Section 6.

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 [26].

The paper presents experimental results for encoding word lists, solutions to a combinatorial problem, and representing the Boolean functions computed by a digital circuit. Both the word list and combinatorial problems can be viewed as representing sets, where each set element is a sequence over an alphabet of symbols. The paper presents a quantitative model for how DDs scale when representing such sets, according to the encoding method used and the DD type. This model further validates our finding that the combination of a one-hot encoding of the symbols, plus a CBDD, CZDD, or ZDD representation of the set yields the most compact form.

The paper concludes with a discussion of the merits of chaining and possible extensions. This paper is an extended version of [8]. As a supplement to this paper, we have set up a web page [9] containing additional material, including links to all source code, copies of all benchmark data, and auxilliary information.

2 Related Work

Over the years, many variants to BDDs have been proposed. One property most variants have preserved is that the reduced version of the decision diagram serves as a *canonical form*. That is, it provides a unique representation of the given function. All the variants we discuss in this paper, including ours, guarantee this property.

Some variants have focused on the interpretation assigned to a node in the graph. For example, Drechsler and Becker [13] allow different decompositions of the variables, in addition to the standard Shannon expansion of BDDs [5]. Others have extended decision diagrams to represent non-Boolean functions [2, 16].

Minato's work on zero-supressed binary decision diagrams [21] was perhaps the first to find a new interpretation for *level-skipping* edges—those that span nonadjacent levels in the variable ordering. His zero-suppression rule can be remarkably effective when encoding very *sparse* sets of binary strings of some fixed length n. Here, sparseness refers to two properties:

- The total set size is much smaller than the maximum of 2^n strings.
- The set elements tend to have many 0s in their strings.

Our experimental results in Section 9 demonstrate the advantages of ZDDs over BDDs when representing such sets. Our analysis in Section 10 extends these empirical observations with

a model quantifying this advantage. Our goal in this work is to be able to harness the relative strengths of both BDDs and ZDDs in a unified framework.

Other recent work has found other methods to unify BDDs and ZDDs. van Dijk and his colleagues devised a hybrid of BDDs and ZDDs they call *tagged BDDs* [12]. 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. We provide some additional comparisons to their approach in Section 12, where we consider possible extensions to the work presented here.

Babar and his colleagues devised a technique for labeling the edges in a DD with four different edge types and then introducing a set of reduction rules that assign the most advantageous edge type for each local context [1]. Their edge types can interpret level-skipping edges as *don't-care* or *zero-suppressed* with respect to the skipped variables, corresponding to BDDs and ZDDs, respectively. They also introduce a new interpretation of these variables as *one-suppressed*. (The fourth edge type is for an edge between adjacent levels.) Their Edge-Specified-Reduction BDDs (ESRBDDs) have the property that they are guaranteed to be no larger than the BDD or ZDD representation of a function. Moreover, relative to BDDs and ZDDs, they only require adding two bits of information for each edge, rather than adding an extra integer index to each node (as in our work) or each edge (as in tagged BDDs). As will be discussed in Section 12, the introduction of the one-suppressed rule allows ESRBDDs to be more compact than all other considered variants (BDDs, ZDDs, their chained counterparts, and tagged BDDs) under some conditions. In their publication, they cite the development of an associated set of algorithms for generating and manipulating ESRBDDs as future work. It therefore remains to be seen how useful this variant of BDDs will be in practice.

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 $l < l'$.

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

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

We refer to these as graphs 2a, 2b, and 2c, respectively. In our illustrations, nonterminal nodes are shown as circles labeled by their levels, and terminal nodes are shown as squares labeled by their values. The *lo* edge from each node is shown as a dashed line, and the *hi* edge is shown as a solid line. To simplify the illustrations, we omit leaf **0** and all branches to it.

Graph 2a 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 $l+1$. (This is similar to

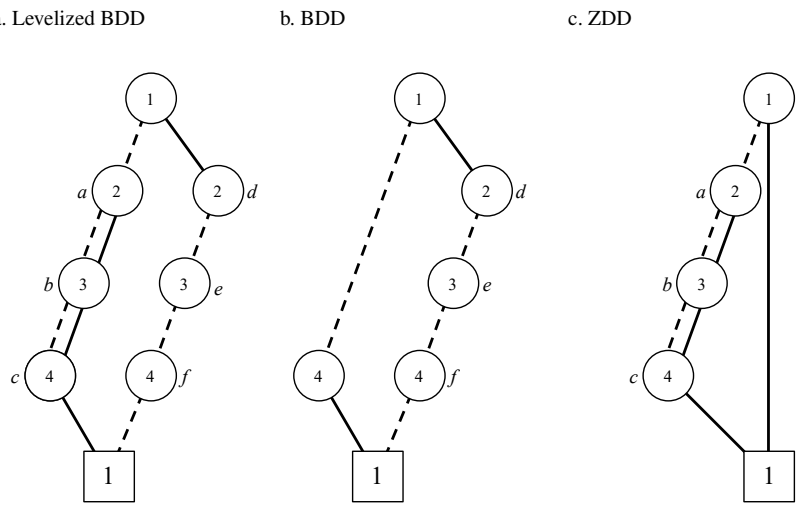


Fig. 2 Reductions in BDDs and ZDDs. Each reduces the representation size with edges between nonconsecutive levels

the *quasi-reduced* form described by Knuth [18], except that his representation only allows edges to leaf **0** from nodes at level n.)

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 2a 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 to 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 2b shows a 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 2a. For ZDDs, a level skipping edge encodes a ZERO chain. Thus, graph 2c 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.

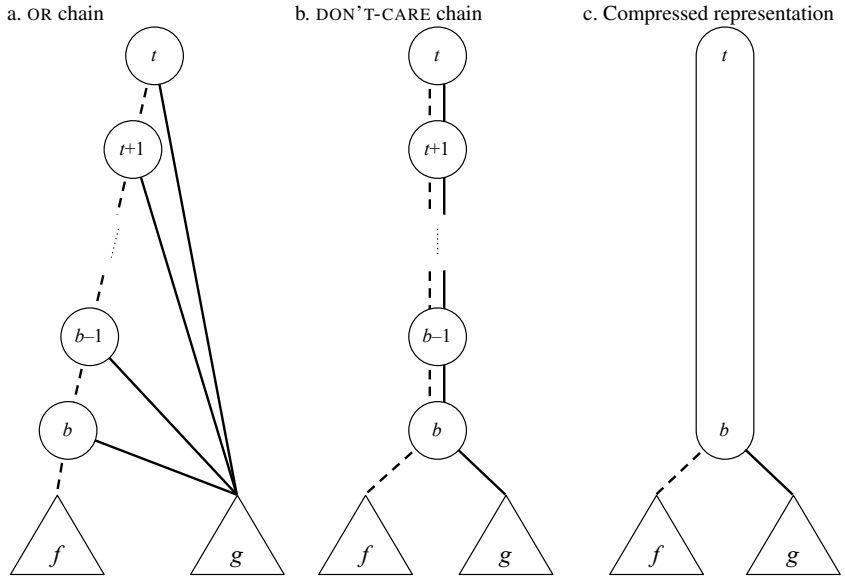


Fig. 3 Chain patterns. These patterns remain after BDD reduction (a), and ZDD reduction (b), but can be represented in compressed form (c).

4 Chain Patterns and Reductions

Fig. 3 shows the general form of OR and DON'T-CARE chains, as were illustrated in the example 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. 3c. That is, every nonleaf node has an associated pair of levels $t:b$, such that $1 \leq t \leq b \leq n$. In our illustrations, nodes are labeled by both values t and b when these are distinct, and by a single level when they are not. We draw a node in elongated form to span from its top level t to its bottom level b. In a *chain-reduced ordered binary decision diagram* (CBDD), such a node encodes the OR chain pattern shown in Fig. 3a, while in a *chain-reduced zero-suppressed binary decision diagram* (CZDD), such a node encodes the DON'T-CARE chain pattern shown in Fig. 3b.

Fig. 4 shows the effect of chain reduction for the example function, 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

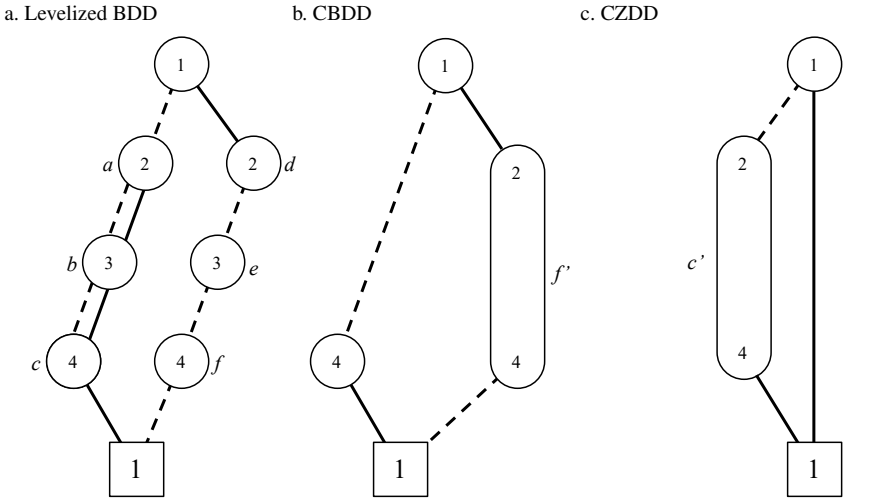


Fig. 4 Chain Reduction Examples. Each now reduces both chain types.

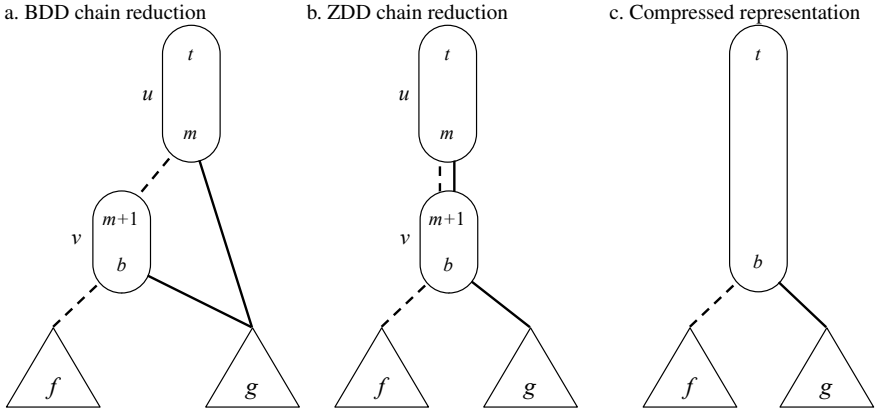


Fig. 5 Chain Reduction Cases. These cases define how chain reduction can be applied to BDDs (a) and ZDDs to obtain the single chain node (c).

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.

5 Generating CBDDs and CZDDs

To generalize from these examples, let us denote a node of the form illustrated in Fig. 3c with the modified if-then-else notation $\langle t : b \rightarrow g, f \rangle$. 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 $u = \langle t : m \rightarrow g, v \rangle$ and $v = \langle m+1 : b \rightarrow g, f \rangle$ (illustrated in Fig. 5a) by the single node $\langle t : b \rightarrow g, f \rangle$ (illustrated in Fig. 5c).

Since reduced BDDs form canonical representations of Boolean functions, and repeated application of the chain compression rule of Fig. 5a will only serve to merge all the nodes in each OR chain, one can readily see that reduced CBDDs also provide canonical representations.

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 $u = \langle t : m \rightarrow v, v \rangle$ and $v = \langle m+1 : b \rightarrow g, f \rangle$ (illustrated in Fig. 5b) is replaced by the single node $\langle t : b \rightarrow g, f \rangle$ (illustrated in Fig. 5c). Again, one can readily see that reduced CZDDs provide canonical representations of Boolean functions.

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.

6 Size Ratio Bounds

The reduction rules for CBDDs and CZDDs (Fig. 5) allow 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 (possibly unreduced) 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.

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. 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 $3k+2$ 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. To construct the graphs for different values of k, the graph pattern enclosed in the diagonal boxes can be replicated as many times as are needed.

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 (possibly unreduced) representation G' as a CZDD. Consider each edge of 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. ZDD/CZDD representation b. CBDD representation

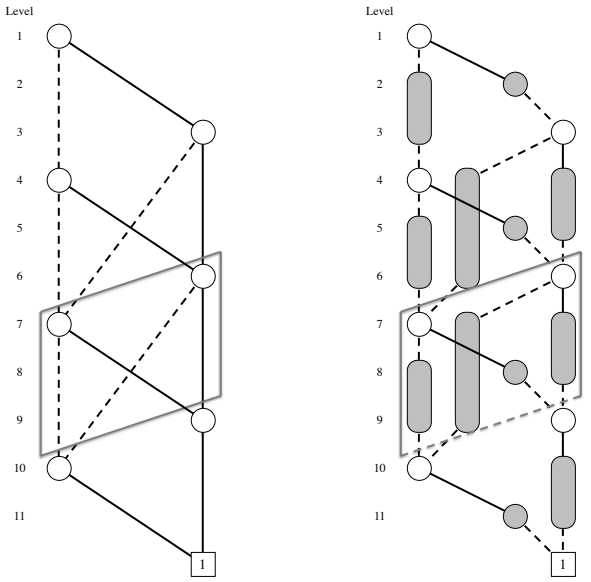


Fig. 6 Worst case example for effectiveness of CBDD compression. The implicit ZERO chains in the ZDD (a) must be explicitly encoded in the CBDD (b)), increasing its size by a factor of 3.

This bound is also tight—Fig. 7 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 representation (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. To construct the graphs for different values of k, the graph pattern enclosed in the diagonal boxes can be replicated as many times as are needed.

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. 3a), where g is something other than $\mathbf{0}$, cannot be directly encoded with a constant number of CZDD nodes.

7 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 [5, 7]. 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

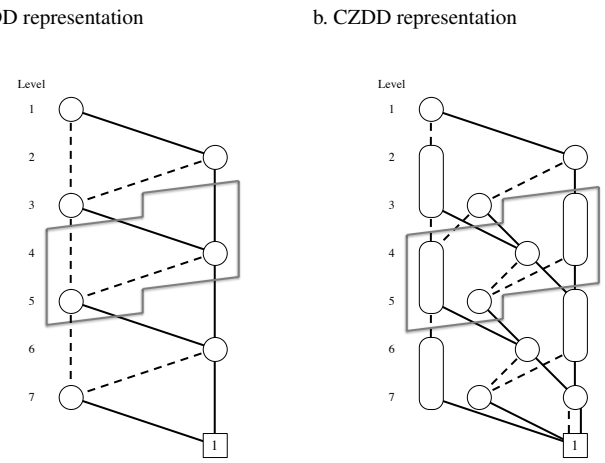


Fig. 7 Worst case example for effectiveness of CZDD compression. The nodes in the BDD (a) must be duplicated to encode the incoming DON'T-CARE chains (b), increasing the size by a factor of 2.

$k = 2$, 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.

7.1 APPLY operation for CBDDs

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

$$t = \min_{1 \le i \le k} t_i \tag{8}$$

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

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	$t_i \le b < b_i$	$\langle b+1 : b_i \to g_i, f_i \rangle$	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.

As an example of the splitting and cofactoring rules, Fig. 8 shows the series of recursive calls generated when applying the OR operation to the arguments indicated by graphs A and B at the top of the figure. The recursive steps proceed as follows. Here we describe them in a breadth-first manner, although the actual control flow would be depth first.

1. The initial recursion has $t = b = 1$, generating arguments A0 and B0 as the *lo* cofactors of A and B, and A1 and B1 as the *hi* cofactors. Observe how generating B0 invoked Case 3 to generate a new node with $t = 2$ and $b = 4$.
2. A1 and B1 form a terminal case, but the recursion for A0 and B0 has $t = 2$ and $b = 3$, yielding *lo* cofactors A00 and B00 and *hi* cofactors A01 and B01. Again observe how generating B00 invoked Case 3 to generate a new node with $t = b = 4$.
3. A01 and B01 also form a terminal case, but a final recursive call is required for A00 and B00 to generate *lo* cofactors A000 and B000, and *hi* cofactors A001 and B001.

Observe that in the recursive steps for this example, no splitting was ever required at level 3. The APPLY operation can exploit chaining to span a range of levels while splitting.

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. In Case 2 of the equation, b' denotes the bottom level of node u_0 (and therefore $b < b'$), and w_0 denotes $lo(u_0)$.

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

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.

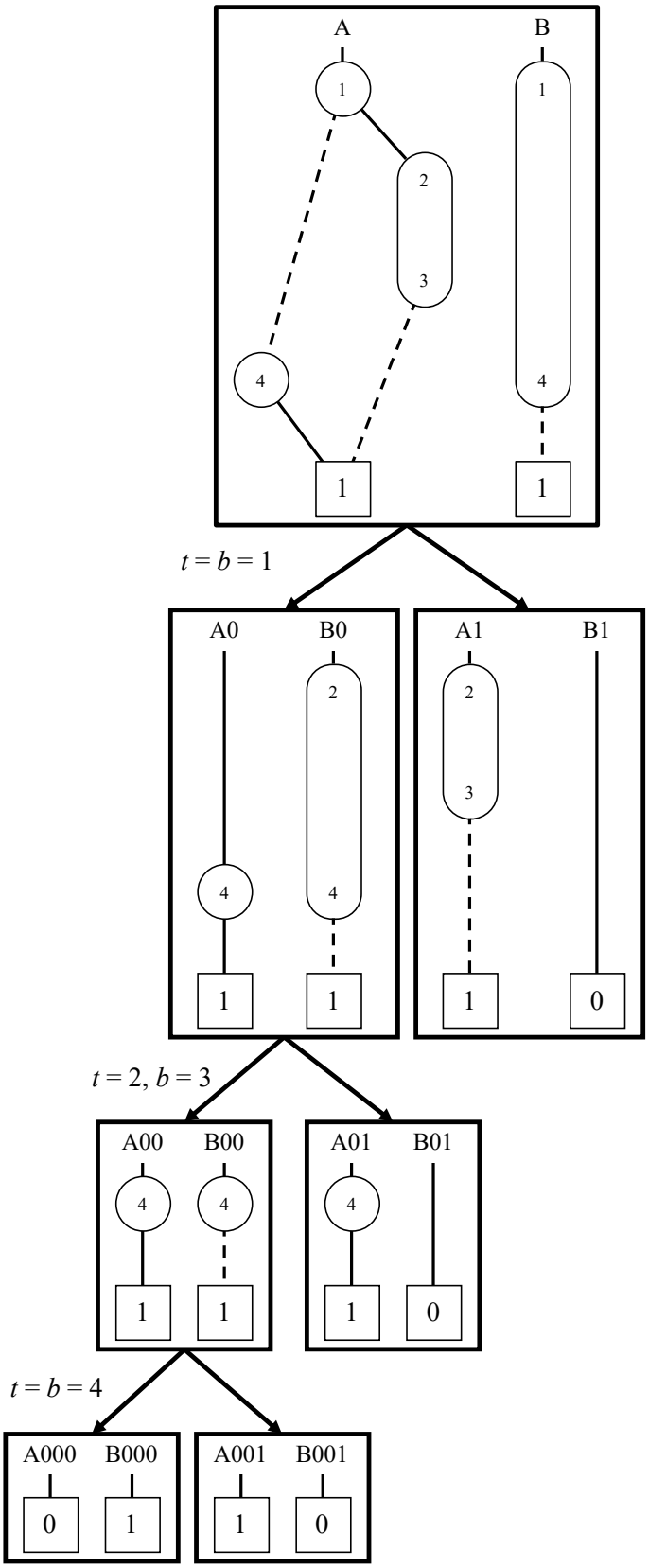


Fig. 8 Recursive calls in example APPLY **operation.** The goal is to compute the OR of arguments A and B. They are recursively split until terminal cases are reached.

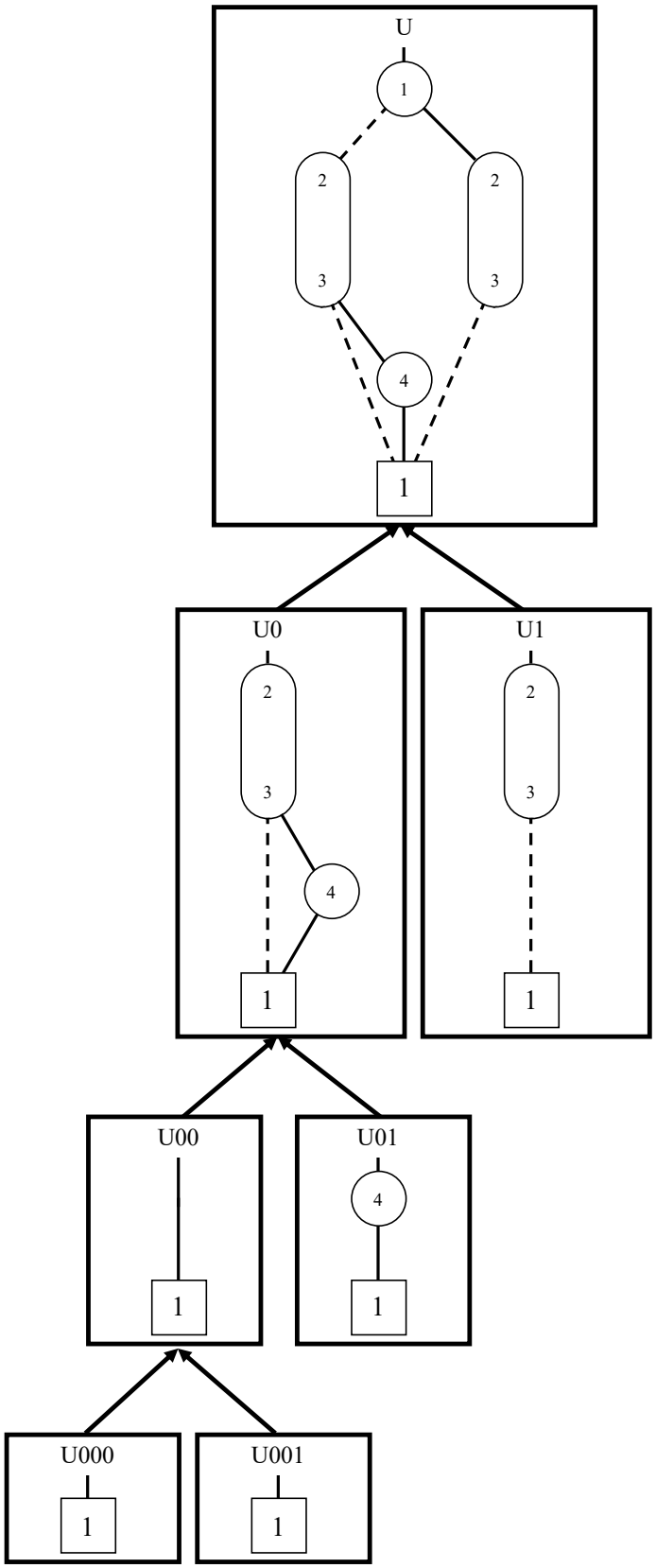


Fig. 9 Construction of result CBDD U as recursive calls in example APPLY **operation return.** The diagram flows from the bottom to the top.

As an example of the combining rules, Fig. 9 shows how the recursive calls diagrammed in Fig. 8 would construct result CBDD U as they return. The construction flows from the bottom to the top of the diagram, inverting the flow of the recursive calls.

1. The lowest level recursive calls yield constant leaf **0** for results U000 and U001.
2. These are combined with by the don't-care reduction (Case 1) to yield leaf **0** for result U00. Meanwhile, the terminal case for arguments A01 and B01 yielded U01 equal to A01.
3. Results U00 and U01 are combined to generate result U0. Since the recursive call had $t = 2$ and $b = 3$, the resulting root node spans these two levels. Meanwhile, the terminal case for arguments A1 and B1 yielded U1 equal to A1.
4. The final result U is generated by combining U0 and U1 with a node having $t = b = 1$.

Observe that the generated result U makes use of the general form of an OR chain—the node on the left has a nonzero value for its *hi* child.

7.2 APPLY operation for CZDDs

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:

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

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

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	$t_i \le b < b_i$	$\langle b+1 : b_i \rightarrow g_i, f_i \rangle$	$\langle b+1 : b_i \rightarrow g_i, f_i \rangle$

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 (11), these are combined to form a result node u, defined as follows. Case 3 of this equation covers the case where $u_1 = u_0$. For

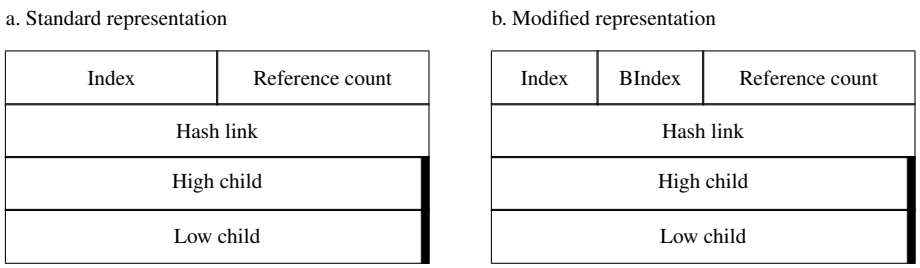


Fig. 10 Data structures for representing DD nodes in the standard implementation of CUDD (a) and modified for chaining (b)

this case, b' denotes the bottom level for the two nodes (and therefore $b < b'$), and w_1 and w_0 denote their children.

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

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.

8 Implementation

We implemented both CBDDs and CZDDs by modifying version 3.0.0 of the CUDD BDD package [26]. We chose CUDD because it already provides implementations of both BDDs and ZDDs, and it has been highly optimized. Fig. 10a illustrates the representation of a node in CUDD, when compiled for 64-bit execution. Each node requires 32 bytes: four bytes for an index, which is translated via a table to indicate the variable level (to support dynamic variable ordering [25]), four bytes for a reference count to support garbage collection, and three eight-byte pointers. One pointer is used to create a singly-linked list of nodes for each bucket in the hash table implementing the unique table [3]. The other two point to the two children. As indicated by the black rectangles on the right-hand edges of these pointers, the low-order bits are used to support *complement edges* [3,23], indicating a reference to either a function or its complement.

Fig. 10b shows our modifications to support chaining. We simply split the index field into two fields, each having two bytes, for top and bottom indices. These are used to determine the top and bottom variable levels for the node. Two bytes is enough to encode 65,535 levels, and so this reduction from four bytes to two imposes no real limitation. Overall, we see that there is no storage penalty for the generalization to a chained form.

Dictionary type	Words	Radix	Length	One-hot variables	Binary variables
Compact word list	235,886	54	24	1,296	144
ASCII word list	235,886	129	24	3,096	192
Compact password list	979,247	80	32	2,560	192
ASCII password list	979,247	129	32	4,128	256

Table 1 Characteristics of dictionary benchmarks

Incorporating chaining required modifications to many parts of the code, including how node keys are generated for the unique table and computed cache, how reductions are performed, and how Boolean operations are applied to functions. In addition, many of the commonly used functions, such as computing the number of satisfying solutions to a function, computing its support, and enumerating a minterm representation of a function required modifications. Most significantly, none of these changes are visible to applications making use of the CUDD application-program interface (API). The representation of a node is not exported to the API, and we made no changes to any of the API function signatures.

The complement edges used by CUDD when representing BDDs can reduce the size of a BDD by a factor of up to two. Such reduction could invalidate the size ratio bounds derived in (5) and (6), and generally make the experimental results more difficult to interpret. For our experimental evaluation, we therefore use a representation based on CUDD's support for *Algebraic Decision Diagrams* (ADDs) [2]. ADDs generalize BDDs by allowing arbitrary leaf values. Restricting the leaf values to 0 and 1 yields conventional BDDs without complement edges. We revisit the use of complement edges in Section 9.5.

9 Experimental Results

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 (I7-7700K) processor with 32 GB of memory running the Macintosh macOS operating system, version 10.13.6.

9.1 Encoding a Dictionary

As has been observed [18], 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 possible symbol is assigned a unique position, and the symbol is represented with a one in this position and 0s 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 padding the shorter words with a special "null" symbol.

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

Table 2 Node counts and ratios of node counts for dictionary benchmarks

Table 1 shows the characteristics of eight Boolean functions derived from two word lists to allow comparisons of different encoding techniques and representations. Both lists are avaiable at the supplementary website [9] The first list is a set of English words in the file `/usr/share/dict/words` found on Macintosh systems and used in early spell checkers. 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.

As Table 1 shows, 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 of the words [14] and then generated Boolean formulas via a depth-first traversal of the Trie.

Table 2 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). 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

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

Table 3 Impact of chaining on effort required to generate DD representations of word lists

of ZDDs (and CZDDs) to use very large, sparse encodings of values. By using chaining, CBDDs can also take advantage of this property.

We further quantify the impact of the encoding scheme (binary vs. one-hot) and the decision-diagram type for representing dictionaries in Section 10.

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. Table 3 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 low-level 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 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.

9.2 The 15-Queens Problem

A second set of benchmarks involves representing all possible solutions to the N-queens problem [22] as a Boolean function. This problem attempts to place N queens on an $N \times N$ 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 has a Boolean variable for each square. A *binary* encoding has $\lceil \log_2 15 \rceil = 4$ variables for each row, encoding the position of the queen within the row.

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

Table 4 Node counts and ratios of node counts for 15-queens benchmarks

Our most successful approach for encoding the N-queens problem with Boolean operations worked from the bottom row to the top, the details of which are available at the supplementary website [9] At each level, it generated formulas for each square, expressing the conditions under which the column directly below, as well as the diagonals downward to the left and to the right, are unoccupied. For each row, the generated formula imposes the constraint that the row must contain a single queen, and it must not conflict with any of the rows below.

Overall, the formulas require around $11N^2$ Boolean operations for a one-hot encoding, and $(8+\log_2 N)\cdot N^2$ Boolean operations for a binary encoding. Although a strict asymptotic complexity analysis would consider $\log_2 N$ to be a significant term in the count, keep in mind that the largest benchmark we have ever successfully completed has $N = 16$, and that $N = 27$ is the largest value for which the total number of solutions is known [24]. It is therefore reasonable to assume that $\log_2 N$ will not exceed 5 for the forseeable future, and hence the total number of operations is $O(N^2)$ for both one-hot and binary encodings.

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 in using this sequence 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. We did not attempt any other variable orderings. We conjecture that the top-down ordering (or its symmetric counterpart, a bottom-up ordering) is optimal. Even if not, our purpose here is to show the relative performance of different DD implementations, and so the optimality of the ordering is not important.

Table 4 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

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

Table 5 Effect of chaining for ZDD representations of 15-queens benchmarks

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.

We further quantify the impact of the encoding scheme (binary vs. one-hot) and the decision-diagram type for representing solutions to the N-queens problem in Section 10.

Table 5 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, we can see that CZDDs have a significantly lower ($3.81\times$) peak requirement. 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. In the end, the center-first variable ordering does not outperform the more obvious, top-down ordering, but it was beneficial to be able to test it. By using chaining, CZDDs can mitigate the risk of trying a nonoptimal variable order.

By way of comparison, Kunkle, Slavici, and Cooperman [19] also used the N-queens problem to test their BDD implementation performed on a cluster of 64 machines. They used a one-hot encoding, but generated the formula by forming a conjunction of all of the constraints on each single position on the board, requiring N^2 constraints, each with around $4N$ terms, requiring a total of $\theta(N^3)$ Boolean operations. In constructing a representation of the 15-queens problem, their program reached a peak of 917,859,646 nodes, a factor of 5.5 times greater than our peak. Our row-by-row encoding technique requires only $\theta(N^2)$ Boolean operations, and it significantly reduces the peak memory requirement. Minato described a third approach for encoding the N-queens problem, but his method also requires $\theta(N^3)$ operations [22].

9.3 Digital Circuits

BDDs are commonly used in digital circuit design automation, for such tasks as verification, test generation, and circuit synthesis. Their ability to represent functions typically found in digital circuits has been widely studied. It is natural to study the suitability of ZDDs, as well as chained versions of BDDs and ZDDs for these applications.

	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

Table 6 Node counts and ratios of node counts for digital circuit benchmarks

We selected the circuits in the ISCAS '85 benchmark suite [4]. These were originally developed as benchmarks for test generation, but they have also been widely used as benchmarks for BDDs [15,20]. We generated variable orderings for all but the last two benchmarks by traversing the circuit graphs, using the fan-in heuristic of [20]. 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 [17].

Fig. 6 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.

Circuit c6288 has historical interest. Integer multiplication is known to be intractable for BDDs regardless of variable ordering [6]. As a result, this benchmark was long considered out of reach for a BDD representation and therefore typically omitted from many benchmark comparisons. With a modern machine, the benchmark is achievable, requiring a peak of around 16 GB to represent the data structures. On the other hand, we see that these functions contain very few DON'T-CARE or OR chains, and hence all four DD types require around 48 million nodes.

9.4 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 $1.4\times$.

Experienced ZDD users take steps to avoid DON'T-CARE chains, for example, by implementing special algorithms that operate directly on ZDDs, rather than expressing a computation as a sequence of Boolean operations [18,28]. By using chaining, CZDDs reduce the

Metric	Complement edges No	Yes	Ratio
Final size	48,177,349	41,417,996	1.16
Operations	675,645,812	272,783,843	2.48
Time (secs.)	1757	280	6.27

Table 7 Benefits of complement edges for CBDD representations of c6288 benchmark

cost of these chains, enabling users to express their computations at the more abstract level of Boolean expressions, rather than implementing special algorithms.

9.5 Complement Edges

As mentioned earlier, CUDD uses complement edges in its representation of BDDs, and hence our experiments showed the results for ADDs to avoid confounding factors in our experimental evaluation. We also implemented CBDDs with complement edges by modifying CUDD's implementation of BDDs. The standard rules for deciding how to canonicalize complement edges [3] can be used without modification with CBDDs. Interestingly, these rules imply that there are no ZERO chains in our CBDDs, because the canonicalization forbids having a *hi* pointer to leaf **0**. Instead, such chains are represented by chain nodes with the *hi* edges pointing to leaf **1**, and with their incoming edges complemented. Although Minato's original paper on ZDDs [23] includes a set of conventions for using complement edges with ZDDs, these are not implemented in CUDD. Thus, we did not attempt this feature in our implementation of CZDDs.

Fig. 7 illustrates the impact of complement edges for CBDDs, based on the c6288 benchmark. Similar results hold without chaining. As can be seen, although complement edges can potentially yield a $2\times$ reduction in the number of nodes, the actual reduction is much smaller ($1.16\times$). However, it can greatly affect the number of operations required ($2.48\times$), since complement edges enable complementing a function in a single step, rather than traversing the graph. (The circuit consists mostly of NOR gates, each requiring a complement operation.) The impact on time ($6.27\times$) is even greater, since the large number of complement operations pollutes the operation cache. The performance gain achieved by complement edges was much less significant for the other benchmarks, in part due to the different mixes of logic gate types.

In terms of implementing and using a BDD package, complement edges provide many sources of complications, bugs, and debugging challenges. Nonetheless, this evaluation indicates that complement edges can, at times, provide a significant performance benefit.

10 Quantitative Model

Both the dictionary and N-queens problem require representing sets of sequences of some length K over an alphabet of R symbols. In the case of a dictionary, the symbols are the characters appearing in the words, along with a null terminator, and each sequence is a word (possibly padded with null symbols). In the case of N-queens, a symbol indicates the position of a queen within a row ($R = N$), and the sequence indicates the placement of the queens in every row. So, for example, `0241C8DBE5F63A79` is the hexadecimal representation of one solution to the 16-queens problem.

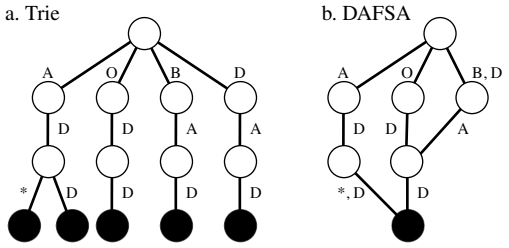


Fig. 11 Trie and DAFSA representations of the set $\{\text{AD}, \text{ADD}, \text{ODD}, \text{BAD}, \text{DAD}\}$

In this section, we formulate a simple, quantitative model for estimating the number of nodes in the decision-diagram representations of such sets, as a function of the encoding method (one-hot or binary) and the decision-diagram type. In comparing this model with the experimental results of Section 9, we find that the total number of nodes in the DD representations for both the word sets and for the N-queens solutions match the model's predictions well. The model provides further insight into the scaling properties of different DD representations.

As notation, for integer x, define $\text{bits}(x) = \lceil \log_2 x \rceil$. So, for example, a binary coding of R symbols requires $\text{bits}(R)$ bits.

Our analysis does not attempt to quantify any advantage CZDDs may have over ZDDs. As we saw with the experimental results, CZDDs do not outperform ZDDs when encoding word sets, except possibly during intermediate points in their construction.

10.1 DAFSA Representations of Sequence Sets

A more direct representation of a finite set of sequences, all of length K is as a reduced finite-state automata representing the set as a language over the symbols in the sequence. This representation is sometime referred to as a *deterministic acyclic finite-state automaton*, or "DAFSA" [11, 10]. As an illustration, Fig. 11 shows Trie (a) [14] and DAFSA (b) representations of the set $\{\text{AD}, \text{ADD}, \text{ODD}, \text{BAD}, \text{DAD}\}$, using symbol '*' as the null terminator. Nonterminal nodes are shown as hollow circles and accepting nodes as solid circles. In a Trie representation, the common prefixes of the words provide a source of sharing by the nodes. With a DAFSA, the suffixes of the words also provide a source of sharing, potentially yielding a considerable reduction in the number of nodes. This sharing of suffixes is precisely the source of compaction that reduced decision diagrams achieve through their sharing of common subgraphs.

We define the *size* of a DAFSA to be the sum of the outdegrees of all of the nodes in the graph, where the outdegree of a node is defined to be the number of symbols for which the node has a successor. As an example, the graph shown in Fig. 11b has size ten, even though it contains eight edges, since all of the labels for an edge count separately when computing the outdegree.

10.2 Encoding Metric

In representing a sequence of symbols as a binary sequence, we must encode each symbol as a sequence of Boolean values. We have considered both one-hot and binary encodings. Assuming the overall ordering of the binary sequence consists of the concatenation of the encodings for the individual symbols, the DD representation of a single sequence then consists of a chain of *selector* function DDs, each yielding 1 for a single symbol. As examples, Fig. 12 shows DD representations of the selector function for the symbol assigned index 9 in a set of 12 symbols.

Define the function $ENC^{T}_{DD}(R)$, to be the average number of nodes per symbol to encode the set consisting of a single sequence as a DD of type *DD* (either BDD, CBDD, or ZDD) using an encoding of type T (either O for one-hot or B for binary). R denotes the size, or *radix*, of the symbol set. That is, if the entire DD has m nodes (not counting the two terminal nodes), and the sequence is of length K, then we compute m/K, and average this across all possible sequences. We assume all symbol values are equally likely.

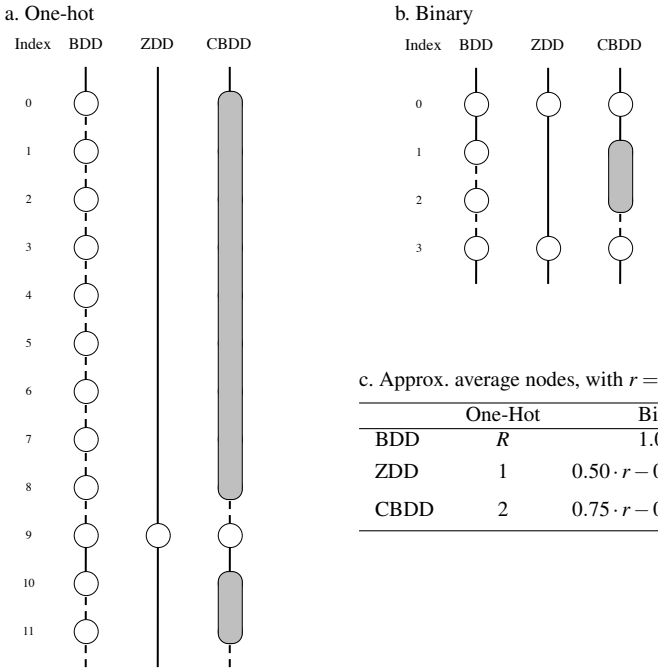
The selector function examples of Fig. 12 provide some insights into these averages. For a one-hot encoding (a), we see that a BDD encoding of the selector requires $R = 12$ nodes, while a ZDD representation requires only 1. The CBDD representation has three nodes, compressing the zero chains above and below the variable with index 9. When we consider the CBDD representation of an entire sequence, we can see that it will contain around $2K$ nodes, since the zero chain at the end of one symbol will merge with the zero chain at the beginning of the next. The encoding metric for one-hot encoding can therefore be characterized as $ENC^{\mathrm{O}}_{\mathrm{BDD}}(R) = R$, $ENC^{\mathrm{O}}_{\mathrm{ZDD}}(R) = 1$, and $ENC^{\mathrm{O}}_{\mathrm{CBDD}}(R) = 2$.

As Fig. 12b illustrates, for a binary encoding, the BDD will require bits(R) variables (4, in this case). We assume the variables are ordered from most significant bit to least, although the DDs will be of the same size if this ordering is inverted. The figure shows the representations for index 9, having binary representation 1001. A BDD will have exactly 4 nodes, but both the ZDD and CBDD representations can have fewer nodes, due to the 0s that occur in the binary representation of the index.

To estimate the values of the encoding metric with binary encoding, let $r = \mathrm{bits}(R)$ and consider the binary sequence of length $r \cdot K$, representing a sequence of K symbols. The BDD representing this sequence will then contain $r \cdot K$ nonterminal nodes. For the ZDD representation, any position in the sequence with value 0 does not require a node. Similarly, for the CBDD representation, any pair of consecutive 0s can be encoded as part of a chained node. Assuming 0s and 1s are equally likely and uniformly distributed, we would expect the DD representations encoding each symbol in a sequence to have, on average, r nodes for a BDD, $0.5r$ nodes for a ZDD, and $0.75r$ nodes for a CBDD.

These simple estimates for the ZDD and CBDD encodings are accurate only for the case of R being a power of two. For other values of R, they are too large, because, in these cases, there will be more 0s than 1s in the sequences, and the 0s are not uniformly distributed. Although the computation of an accurate estimate for all values of R is complex, we can introduce a correction factor for the prevalence of 0s in the most significant bits of the encodings. That is, the most significant bits will be zero, on average, for fraction $2^{r-1}/R$ of the cases. These 0s will reduce the number of ZDD nodes required to represent the selector functions, and they will provide more opportunities for chaining in CBDDs. For both cases, the correction factor is $-0.5(2^r/R - 1)$. When $R = 2^r$, this factor drops to zero, but it can be close to -0.5 when R is close to its lower bound value of $2^{r-1} + 1$.

Fig. 12c summarizes these estimates on the average number of nodes to encode each symbol in the DD representation of a sequence of symbols.



c. Approx. average nodes, with $r = \text{bits}(R)$

	One-Hot	Binary
BDD	R	$1.00 \cdot r$
ZDD	1	$0.50 \cdot r - 0.5\left(\frac{2^r}{R} - 1\right)$
CBDD	2	$0.75 \cdot r - 0.5\left(\frac{2^r}{R} - 1\right)$

Fig. 12 DD representations of a selector function selecting index 9 (binary 1001) from a set of 12 symbols, and their average sizes

When representing a set of sequences as a DD, we can estimate its size to be proportional to the size of its DAFSA representation times the size of the selector functions encoding the DAFSA nodes. But, the DDs representing these selector functions can have additional sharing. For example, a DAFSA node with outdegree $R = 2^r$ would require just $R - 1$ nodes to encode as a BDD, when using a binary encoding, well less than the naive estimate of $R \cdot r$ nodes. Similarly, there can be sharing among the DDs representing different DAFSA nodes within a single level. For a set of sequences S, let DAFSA(S) be the size of its DAFSA representation, and let $DD^T(S)$ be the size of its DD representation for encoding method T and DD type DD.

Define the *sharing ratio* α as

$$\alpha = \frac{ENC^T_{DD}(R) \cdot \text{DAFSA(S)}}{DD^T(S)} \tag{12}$$

That is, α quantifies the degree of sharing among the DD nodes when encoding the DAFSA nodes.

By its definition, we must have $\alpha \geq 1$, since the numerator in (12) represents the size of the DD obtained by encoding every outgoing branch from a DAFSA node as a separate selector function. On the other hand, if we find that α can be much greater than 1.0, or that it varies widely according to the underlying DAFSA, encoding method, or DD type, this would indicate that our quantitative model is too simplistic to capture the scaling properties of DDs.

One-hot	DAFSA	Encoding cost			Sharing ratio		
	size	BDD	CBDD	ZDD	BDD	CBDD	ZDD
Compact word list	411,152	54.00	2.00	1.00	2.29	1.31	1.38
ASCII word list	411,152	129.00	2.00	1.00	2.29	1.31	1.38
Compact password list	1,600,190	80.00	2.00	1.00	2.60	1.38	1.42
ASCII password list	1,600,190	129.00	2.00	1.00	2.61	1.38	1.42
Binary	DAFSA	Encoding cost			Sharing ratio		
	size	BDD	CBDD	ZDD	BDD	CBDD	ZDD
Compact word list	411,152	6.00	4.41	2.91	2.21	1.80	1.65
ASCII word list	411,152	8.00	5.51	3.51	2.25	1.77	1.69
Compact password list	1,600,190	7.00	4.95	3.20	2.53	2.20	2.04
ASCII password list	1,600,190	8.00	5.51	3.51	2.59	2.05	1.95

Table 8 Sharing ratios for dictionary benchmarks

N	8	9	10	11	12	13	14	15
DAFSA size	452	1,639	3,806	12,985	58,094	264,409	1,190,178	6,380,499

Table 9 DAFSA sizes for N-queens solutions

10.3 Experimental Evaluation of Model

Table 8 shows the sharing ratios for the dictionary benchmarks. These range between 1.31 and 2.61, a fairly tight range considering the 16-fold range in the DD sizes of Table 2. For the one-hot encodings, we see that BDDs have a significantly higher sharing ratio than ZDDs. The long chains of BDD nodes in a one-hot encoding (e.g., Fig. 12a) provide many opportunities for sharing among and between DAFSA node encodings. The sharing ratios of ZDDs are slightly higher than those for CBDDs. The CBDD nodes encoding zero chains can be shared only if they match on both their top and bottom indices. The combination of CBDDs requiring two nodes per symbol, while ZDDs require only one, and the difference in sharing ratios accounts for the factor of 2.05–2.10 that we saw of CBDDs to ZDDs in the final column of Table 8.

For the binary encodings, the sharing ratios seem generally comparable, ranging between 1.65 and 2.59, with BDDs having slightly higher ratios than CBDDs, and these having slightly higher ratios than ZDDs. These higher ratios counteract the (already small) predicted relative advantages of ZDDs over CBDDs, and CBDDs over BDDs.

Overall, we can see that our model explains why the combination of a one-hot encoding and the use of ZDDs, CZDDs, or CBDDs yields the most compact DD representations for the sample word sets—this combination provides constant scaling with respect to the DAFSA size, independent of the radix R. Using a binary encoding causes the DDs to scale by $\log R$, with only a small advantage of ZDDs over CBDDs and CBDDs over BDDs. As predicted by the model, and confirmed by our experimental results, BDDs do not scale well when a one-hot encoding is used.

For the N-queens problem, we computed sharing ratios for all values of N between 8 and 15, and for all six combinations of encoding method and DD type. The DAFSA sizes are shown in Table 9. Note the very wide range of DAFSA sizes—over three orders of magnitude. Fig. 13 shows plots of the resulting sharing ratios. For one-hot encodings, the BDD representations have significantly higher sharing ratios than ZDDs or CZDDs, because their long chains provide many opportunities for sharing. We see a somewhat higher ratio for ZDDs than for CBDDs, with the result that the BDDs requires between 2.06 and 2.20

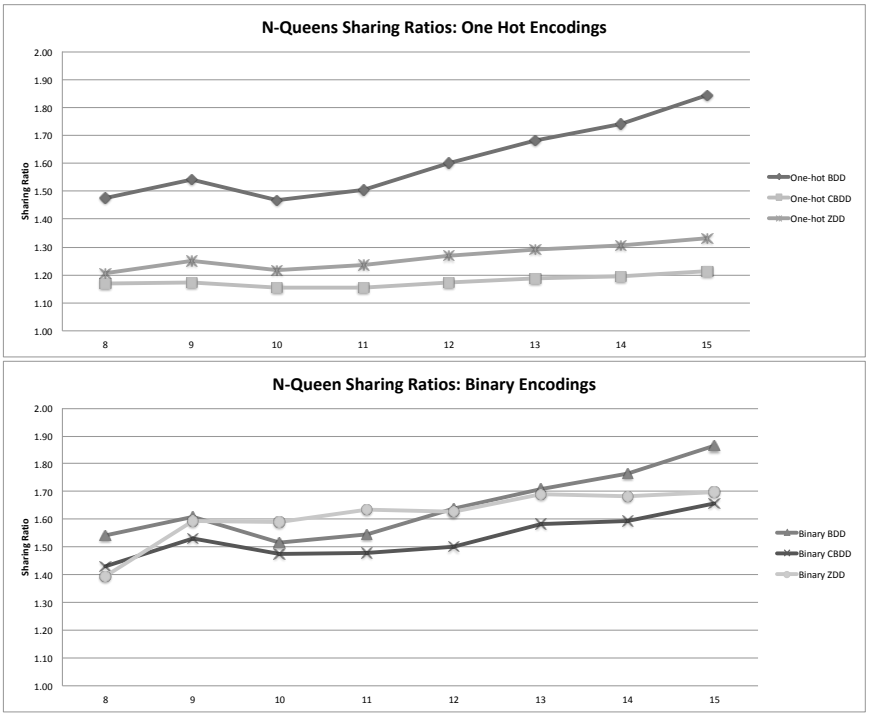


Fig. 13 Sharing ratios for N-queens benchmarks

times more nodes to represent than the ZDDs, rather than the factor of 2.00 predicted by our model. For the binary encodings, the three DD types yield similar sharing ratios.

Our simple model of DD size shows the impact the encoding method and the DD type have on their sizes. For representing a set of sequences, the fundamental structure is a DAFSA, encoding all of the sequences with a sharing of prefixes and suffixes. Using a one-hot encoding with either a ZDD or a CBDD, the resulting DD will be within a small constant factor of the DAFSA size, with the CBDD being around twice the size of the ZDD. This shows that our worst-case bound of 3 is pessimistic for this class of applications. Using a one-hot encoding with a BDD representation is clearly inferior, causing a scaling by a factor of the radix R. Using a binary encoding of the symbols yields DDs that scale as the log of the radix, relative to the DAFSA size. The overall sizes scale as ZDD $<$ CBDD $<$ BDD, but these are by small, constant factors.

Overall, the best choice for minimizing DD size is to use a one-hot encoding with ZDDs, or possibly CZDDs to reduce the total number of operations. CBDDs with a one-hot encoding are also viable choices.

11 Conclusions

We have shown that incorporating chain compression into a decision diagram representation can yield DDs that exploit the forms of compaction found in both BDDs and ZDDs. This allows a user to test different encoding schemes and variable orderings with less risk of a

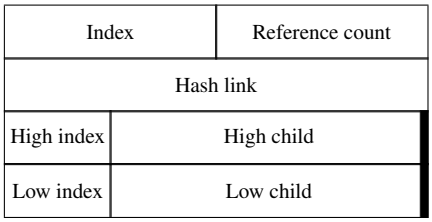


Fig. 14 Possible data structure for representing tagged BDD nodes

blow-up due to a nonoptimal choice of DD type. Our experimental results and our analytic model show that CBDDs can achieve a level of compaction within a constant factor of ZDDs, while CZDDs can avoid some of the performance problems that arise when using ZDDs.

12 Comparisons and Further Work

Our modifications to CUDD to support chaining were only sufficient to evaluate the basic concepts. Fully integrating these changes into such a complex software package would require significantly more effort. Perhaps the most challenging would be to implement dynamic variable ordering [25]. The basic principles that enable dynamic variable ordering for BDDs also hold for both CBDDs and CZDDs. That is, an exchange of variables at levels l and $l+1$ can be performed without modifying any of the nodes with levels less than l and without modifying any external node pointers. Special consideration must be given to nodes with levels $t:b$, such that either $t = l+1$ or $b = l$. Many low-level details and heuristics of the implementation would need to be altered to enable dynamic variable ordering to work well.

The tagged BDD representation of van Dijk and his colleagues [12] has the intriguing property that it compresses both DON'T-CARE and ZERO chains, and so the tagged-BDD representation of a function will never have more nodes than either its BDD or its ZDD representation. Their approach associates a variable level with each edge in the graph, indicating a position where the edge transitions from a DON'T-CARE to a ZERO chain. They describe a modified version of the APPLY algorithm to take account of these transitions. Naturally, the algorithms are more complex than the standard BDD operations, and indeed somewhat more complex than the ones we have presented for chained DDs, but the underlying principles are comparable.

Fig. 14 shows a possible data structure representation of a node in a tagged BDD. Rather than allocating an eight byte pointer for each child, it reserves two of those bytes for an index, and the remaining six bytes for the pointer. Given that contemporary processors limit their virtual address spaces to 2^{48} bits (enough to reference over 281 terabytes), this structure would not impose a limit on the sizes of DDs that could be represented. On the other hand, it would greatly complicate the data-structure references, including in many portions of typical application code. Since node pointers (including the low-order bit for complementation) are referenced directly in application code, reserving bytes in these pointers for variable indices would lead to many incompatibilities with existing code. Nonetheless, this extension should be given careful consideration.

We did not attempt a comparative evaluation of our benchmarks for tagged BDDs, but the paper by Babar, et al., includes some comparisons between tagged BDDs and other representations [1]. On dictionary benchmarks similar to ours, tagged BDDs exactly match the node counts of ZDDs (and CZDDs). For circuit benchmarks similar to ours (same circuits, but different variable orderings), the TBDDs were slightly more compact than BDDs (or CBDDs) for some cases, but never more than by a few percent. It seems, therefore, that tagged BDDs mainly provide a way to exploit the best aspects of ZDDs and BDDs, rather than fundamentally improving on either.

By contrast, the experimental results by Babar, et al., show that ESRBDDs can significantly outperform all other representations, in terms of node counts, for the dictionary benchmarks when using binary encodings. In one of their benchmarks, the ESRBDD representation has 2,410,589 nodes, versus 3,532,847 for the next-best representation as a ZDD, a factor of 1.47$\times$ smaller. The one-suppression rule of ESRBDDs allows them to have fewer nodes when representing selector functions for binary patterns containing sequences of consecutive 1s as well as sequences of consecutive 0s. At the supplementary website, we present a derivation that shows that the ESRBDD encoding of a binary sequence of length r has, on average $r/3$, nodes, assuming 0s and 1s are equally likely and uniformly distributed [9]. This is a fundamental improvement on the averages derived for BDDs (r), CBDDs ($3r/4$), and ZDDs ($r/2$). The observed factor of 1.47 for ESRBDDs over ZDDs nearly achieves the ratio of 1.50 that this analysis predicts. This result is also an indication of the predictive power of our quantitative model.

Acknowledgements This work has benefited from conversations with Shin-Ichi Minato and Ofer Strichman. This work was supported, in part, by NSF STARSS grant 1525527.

References

1. Babar, J., Jiang, C., Ciardo, G., Miner, A.: Binary decision diagrams with edge-specified reductions. In: Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 11428, pp. 303–318 (2019)
2. 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 (1993)
3. 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 (1990)
4. 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)
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986)
6. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Transactions on Computers **40**(2), 205–213 (1991)
7. Bryant, R.E.: Binary decision diagrams. In: E.M. Clarke, T.A. Henzinger, H. Veith, R. Bloem (eds.) Handbook of Model Checking. Springer Verlag (2018)
8. Bryant, R.E.: Chain reduction for binary and zero-suppressed decision diagrams. In: Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 10805, pp. 81–98 (2018)
9. Bryant, R.E.: Supplementary material regarding chain reduction for binary and zero-suppressed decision diagrams. `http://www.cs.cmu.edu/~bryant/bdd-chaining.html` (2020)
10. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite state automata and transducers. Computational Linguistics **26**, 3–16 (2000)
11. Daciuk, J., Watson, B.W., Watson, R.E.: Incremental construction of minimal acyclic finite state automata and transducers. In: Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, pp. 48–56 (1998)

12. 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)
13. Drechsler, R., Becker, B.: Ordered Kronecker function decision diagrams. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17**(10), 965–973 (2006)
14. Fredkin, E.M.: Trie memory. Communications of the ACM **3**, 490–500 (1960)
15. 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 (1988)
16. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in Systems Design **10**, 149–169 (1997)
17. Hansen, M., Yalcin, H., Hayes, J.P.: Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. IEEE Design and Test **16**(3), 72–80 (1999)
18. Knuth, D.E.: The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part I. Addison Wesley (2011)
19. Kunkle, D., Slavici, V., Cooperman, G.: Parallel disk-based computation for large, monolithic binary decision diagrams. In: International Workshop on Parallel and Symbolic Computation, pp. 63–72. ACM (2010)
20. 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 (1988)
21. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th ACM/IEEE Design Automation Conference, pp. 272–277 (1993)
22. Minato, S.: Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Academic Publishers (1995)
23. 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 (1990)
24. Preußer, T.B., Engelhardt, M.R.: Putting queens in carry chains, No. 27. Journal of Signal Processing Systems **88**(2), 185–201 (2017)
25. Rudell, R.L.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 139–144 (1993)
26. Somenzi, F.: Efficient manipulation of decision diagrams. International Journal on Software Tools for Technology Transfer **3**(2), 171–181 (2001)
27. Wegener, I.: Branching Programs and Binary Decision Diagrams: Theory and Applications. SIAM (2000)
28. Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. zero-suppressed BDDs for CTL symbolic model checking of Petri nets. In: Formal Methods in Computer-Aided Design, *Lecture Notes in Computer Science*, vol. 1166, pp. 435–449 (1996)