DenseZDD: A Compact and Fast Index for Families of Sets

Shuhei Denzumi[1], Jun Kawahara[2], Koji Tsuda[3,4], Hiroki Arimura[1], Shin-ichi Minato[1,4], and Kunihiko Sadakane[5]

[1)]Graduate School of IST, Hokkaido University, Japan
[2)]Nara Institute of Science and Technology (NAIST), Japan
[3)]National Institute of Advanced Industrial Science and Technology (AIST), Japan
[4)]ERATO MINATO Discrete Structure Manipulation System Project, JST, Japan
[5)]National Institute of Informatics (NII), Japan
{denzumi,arim,minato}@ist.hokudai.ac.jp,jkawahara@is.naist.jp,
koji.tsuda@aist.go.jp,sada@nii.ac.jp

Abstract. In many real-life problems, we are often faced with manipulating families of sets. Manipulation of large-scale set families is one of the important fundamental techniques for web information retrieval, integration, and mining. For this purpose, a special type of *binary decision diagrams (BDDs)*, called *Zero-suppressed BDDs (ZDDs)*, is used. However, current techniques for storing ZDDs require a huge amount of memory and membership operations are slow. This paper introduces DenseZDD, a compressed index for static ZDDs. Our technique not only indexes set families compactly but also executes fast membership operations. We also propose a hybrid method of DenseZDD and ordinary ZDDs to allow for dynamic indices.

1 Introduction

Binary Decision Diagrams (BDD) [1] are a graph-based representation of Boolean functions and widely used in VLSI logic design and verification. A BDD is constructed reducing a binary decision tree, which represents a decision making process through the input variables. If we fix the order of the input variables and apply the following two reduction rules, then we obtain a minimal and canonical form for a given Boolean function:

1. Delete all redundant nodes (whose two children are identical) and
2. Merge all equivalent nodes (having the same index and pair of children).

Among unique canonical representations of Boolean functions, BDDs are smaller than others such as CNF, DNF, and truth tables for many classes of functions. BDDs have the following features:

- Boolean functions are uniquely represented like other representations.
- Multiple functions are stored compactly by sharing common subgraphs.
- Fast logical operations are executed on Boolean functions.

Zero-suppressed Binary Decision Diagrams (ZDDs) [8] are variation of traditional BDDs, used to manipulate families of sets. Using ZDDs, we can implicitly enumerate combinatorial item set data and efficiently compute set operations over the ZDDs. In the rest of this section, we use the term BDD to indicate both the original BDD and the ZDD unless specified because any ZDD is regarded as a BDD representing some function.

Though BDDs are more compact than other representations of Boolean function and set families, they are still large; a node of a BDD uses 20 to 30 bytes depending on implementations [9]. BDDs become inefficient if the graph size is too large to be held in memory. Therefore the aim of this paper is to reduce the size (number of bits) used to represent BDDs. We classify implementations of BDDs into three types:

- Dynamic: The BDD can be modified. New nodes can be added to the BDD.
- Static: The BDD cannot be modified. Only query operations are supported.
- Freeze-dried: All the information of the BDD is stored, but it cannot be used before restoration.

Most of the current implementations of BDDs are dynamic. There is previous work on freeze-dried representations of BDDs by Starkey and Bryant [16] and later, by Mateu and Prades-Nebot [7]. Hansen, Rao and Tiedemann [4] developed that a technique to compress BDD and reduce the size of the BDD to 1-2 bits per node. However there is no implementation of BDDs that is specialized for static type.

This paper is the first to propose a static representation of ZDDs, which we call *DenseZDDs*. The size of ZDDs in our representation is much smaller than an existing dynamic representation [9]. Not only compact, DenseZDD supports much faster membership operations than [9]. Experimental results show that DenseZDDs are five times smaller and membership queries are twenty to several hundred times faster, compared to [9]. Note that our technique can be directly applied to compress traditional BDDs too.

2 Preliminaries

Let $e_1, \ldots, e_n$ be items such that $e_1 < e_2 < \cdots < e_n$. Let $S = \{a_1, \ldots, a_c\}$, $c \geq 0$, be a set of items. We denote the *size* of S by $|S| = c$. The empty set is denoted by $\emptyset$. A *family* is a subset of the power set of all items. A finite family F of sets is referred to as a *set family*. [1] The *join* of families F_1 and F_2 is defined as $F_1 \sqcup F_2 = \{ S_1 \cup S_2 \,|\, S_1 \in F_1, S_2 \in F_2 \}$.

In the appendix, we describe existing succinct data structures. The balanced parenthesis sequence (BP), the Fully Indexable Dictionary (FID), and some basic structures used in this paper are reviewed. We also explain operations on the data structures such as $rank_c$, $select_c$, and so on.

[1] In the original ZDD paper by Minato, a set is called a combination, and a set family is called a combinatorial set.

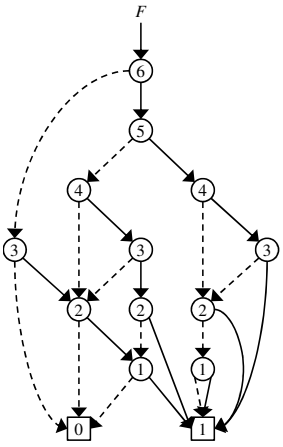

Fig. 1. An example of ZDD.

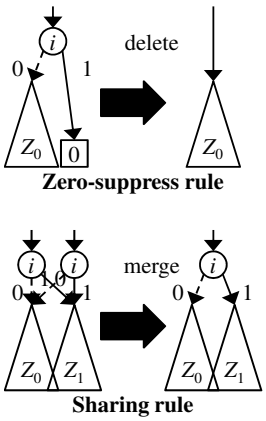

Fig. 2. Reduction rules of ZDDs.

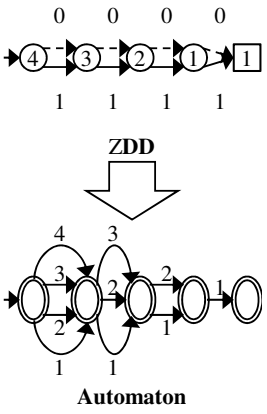

Fig. 3. Worst-case example of a straightforward translation.

2.1 Zero-suppressed Binary Decision Diagrams (ZDDs)

A *zero-suppressed binary decision diagram* (a ZDD) [8] is a variant of a binary decision diagram [1], customized to manipulate finite families of sets.

A ZDDis a directed acyclic graph satisfying the following. A ZDD has two types of nodes, terminal and nonterminal nodes. A *terminal node* v has as attribute a value $value(v) \in \{0, 1\}$, indicating whether it is a *0-terminal node* or a *1-terminal node*, denoted by **0** and **1**, respectively. A *nonterminal node* v has as attributes an integer $index(v) \in \{1, \ldots, n\}$ called the *index*, and two children $zero(v)$ and $one(v)$, called the *0-child* and *1-child*. The edges from nonterminals to their 0-child (1-child resp.) are called *0-edges* (*1-edges* resp.). In the figures, terminal nodes are denoted by squares, and nonterminal nodes are denoted by circles. 0-edges are denoted by dotted arrow, and 1-edges are denoted by solid arrow. We define $triple(v) = \langle index(v), zero(v), one(v) \rangle$, called the *attribute triple* of v. For any nonterminal node v, $index(v)$ is larger than the indices of its children. [2] We define the *size* of the graph, denoted by $|G|$, as the number of its nonterminals.

Definition 1 (set family represented by ZDD). *A ZDD G rooted at a node $v \in V$ represents a finite family of sets $F(v)$ on U_n defined recursively as follows: (1) If v is a terminal node: $F(v) = \{\emptyset\}$ if $value(v) = 1$, and $F(v) = \emptyset$ if $value(v) = 0$. (2) If v is a nonterminal node, then $F(v)$ is the finite family of sets $F(v) = (\{e_{index(v)}\} \sqcup F(one(v))) \cup F(zero(v))$.*

The example in Fig. 1 represents a sets family $F = \{ \{6, 5, 4, 3\}, \{6, 5, 4, 2\}, \{6, 5, 4, 1\}, \{6, 5, 4\}, \{6, 5, 2\}, \{6, 5, 1\}, \{6, 5\}, \{6, 4, 3, 2\}, \{6, 4, 3, 1\}, \{6, 4, 2, 1\},$

[2] In ordinary BDD or ZDD papers, the indices are in ascending order from roots to terminals. For convenience, we employ the opposite ordering in this paper.

Table 1. Main operations supported by ZDD. The first group are the primitive ZDD operations used to implement the others, yet they could have other uses.

$index(v)$	Returns the index of node v.
$zero(v)$	Returns the 0-child of node v.
$one(v)$	Returns the 1-child of node v.
$getnode(i, v_0, v_1)$	Generates (or makes a reference to) a node v with index i and two child nodes $v_0 = zero(v)$ and $v_1 = one(v)$.
$topset(v, i)$	Returns a node with the index i reached by traversing only 0-edges. If such a node does not exist, return the 0-terminal node.
$member(v, S)$	Returns *true* if $S \in F(v)$, and returns *false* otherwise.
$count(v)$	Returns $\|F(v)\|$.
$offset(v, i)$	Returns v such that $F(v) = \{ S \subseteq U_n \mid S \in F, e_i \notin S \}$.
$onset(v, i)$	Returns v such that $F(v) = \{ S \backslash \{e_i\} \subseteq U_n \mid S \in F, e_i \in S \}$.
$apply_\diamond(v_1, v_2)$	Returns v such that $F(v) = F(v_1) \diamond F(v_2)$, for $\diamond \in \{\cup, \cap, \backslash, \oplus\}$.

$\{6, 2, 1\}$, $\{3, 2, 1\}$, $\}$. A set $S = \{a_1, \ldots, a_c\}$ describes a path in the graph G starting from the root. At each nonterminal node, the path continues to the 0-child if $e_i \notin S$ and to the 1-child if $e_i \in S$. The path eventually reaches the 1-terminal (or 0-terminal resp.), indicating that S is accepted (or rejected resp.).

In ZDD, we employ the following two reduction rules to compress the graph: (a) Zero-suppress rule: A nonterminal node whose 1-child is the 0-terminal node. (b) Sharing rule: Two or more nonterminal nodes having the same attribute triple. By applying above rules, we can reduce the graph without changing its semantics. If we apply the two reduction rules as much as possible, then we obtain a canonical form for a given family of sets.

We can reduce the size of ZDDs by using a type of *attributed edges* [11] named *0-element edges*. We have to place a couple of constraints on using 0-element edges to keep the uniqueness of the graphs: (1) Use the 0-terminal node only. (2) Do not use 0-element edges at the 0-edge on each node. Each 1-edge of nonterminal node v has an *$\emptyset$-flag* $empflag(v)$ to implement 0-element edges. If $empflag(v) = 1$, the subgraph pointed by the v's 1-edge includes the empty set $\emptyset$ in the family represented by the subgraph. In the figures in this paper, effective $\emptyset$-flags are denoted as small circles at starting points of 1-edges.

Table 1 summarizes operations of ZDDs. The upper half shows the primitive operations, while the lower half shows other operations which can be implemented by using the primitive operations. The operations $index(v)$, $zero(v)$, $one(v)$, $topset(v, i)$ and $member(v, S)$ do not create new nodes. Therefore they can be done on a static ZDD. Note that the operation $count(v)$ does not create any node; however we need an auxiliary array to memorize which nodes are already visited.

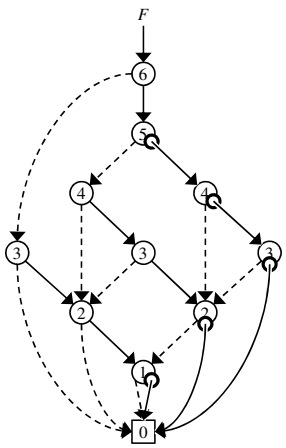

Fig. 4. The ZDD using 0-element edges that is equivalent to the ZDD in Fig. 1.

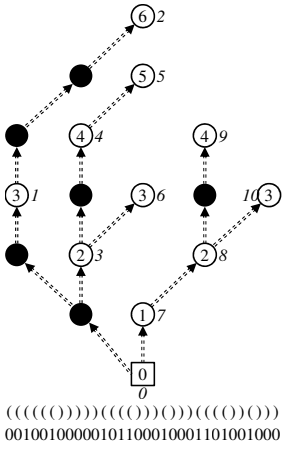

Fig. 5. A zero-edge tree and a dummy node vector obtained from the ZDD in Fig. 4.

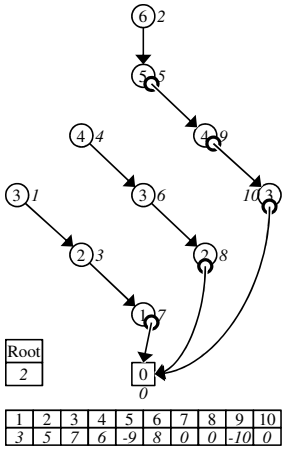

Fig. 6. A one-child array obtained from the ZDD in Fig. 4.

2.2 Problem of existing ZDDs

Let m be the number of nodes of a given ZDD and n be the number of distinct indices of nodes. Existing ZDD implementations have the following problems. First, they require too much memory to represent a ZDD. Second, the $member(v, S)$ operation is too slow, needing $\Theta(n)$ time in the worst case. In practice, the size of query sets is usually much smaller than n, and so an $\mathcal{O}(|S|)$ time algorithm is desirable. However it must be impossible to attain this in the current implementation [9] because the $member(v, S)$ operation is implemented by using the $zero(v)$ operation repeatedly.

For example, we traverse 0-edges 255 times when we search $S = \{e_1\}$ on the ZDD for $F = \{\{e_1\}, \ldots, \{e_{256}\}\}$. If we translate a ZDD to an equivalent automaton by using an array to store pointers (See Fig. 3), we can execute searching in $\mathcal{O}(|S|)$ time. ZDD nodes correspond to labeled edges in the automaton. However, the size of such automaton via a straightforward translation can be $\Theta(n)$ times larger than the original ZDD [2] in the worst case. Therefore, we want to perform $member(v, S)$ operations in $O(|S|)$ time on ZDDs.

Minato proposed Z-Skip Links [10] to accelerate the traversal of ZDDs of large-scale sparse datasets. Their method adds one link per node to skip nodes that are concatenated by 0-edges. Therefore, memory requirement can not be smaller than original ZDDs. Z-Skip-Links makes the membership operations much faster than using conventional ZDD operations when handling large-scale sparse datasets. However, the computation time is probabilistically analyzed only for average case.

3 Data structure

In this section, we show our data structure *DenseZDD* which solves the two problems as described in Section 2.2. We obtain the following results.

Theorem 1. *A ZDD with m nodes on n items can be stored in $2u + m \log m + 3m + o(u)$ bits so that the primitive operations except getnode(i, v_0, v_1) are done in constant time, where u is the size of the ZDD that removes the zero-suppress rule only for nodes pointed by 0-edges. In other words, u is the size of the ZDD with dummy nodes that is described bellow. The getnode(i, v_0, v_1) operation is done in $\mathcal{O}(\log m)$ time.*

Theorem 2. *A ZDD with m nodes on n items can be stored in $\mathcal{O}(m(\log m + \log n))$ bits so that the primitive operations are done in $\mathcal{O}(\log m)$ time except getnode(i, v_0, v_1). The getnode(i, v_0, v_1) operation is done in $\mathcal{O}(\log^2 m)$ time.*

3.1 DenseZDD

A DenseZDD $DZ = \langle U, M, I \rangle$ is composed of three data structures: a zero-edge tree U, a dummy node vector M, and a one-child array I.

Zero-edge tree : The spanning tree of ZDD G formed by the 0-edges is called the *zero-edge tree* of G and denoted by T_Z. In a zero-edge tree, all 0-edges are reversed and the 0-terminal node become the root of the tree. These are equivalent to the preorder rank of each node. Zero-edge trees are based on the same idea as left or right trees by Maruyama *et al.* [6].

An important difference between our structure and theirs is the existence of *dummy nodes*. We call nodes in the original ZDD as *real nodes*. We use the zero-edge tree with dummy nodes, denoted by T'_Z. We create dummy nodes on each 0-edge to guarantee that the depth of every real node v in the zero-edge tree equals $index(v)$. We define the depth of the 0-terminal node, the root of this tree, is 0. Let U be the BP of T'_Z. The length of U is $\mathcal{O}(mn)$ because we create $n - 1$ dummy nodes for one real node in the worst case. An example of a zero-edge tree and its BP are shown in Fig. 5. Black circles are dummy nodes and the number next to each node is its preorder rank. The 0-terminal node is ignored in the BP because we know the root of a zero-edge tree is always it.

Dummy node vector : A bit vector of the same length as U is used to distinguish dummy nodes and real nodes. We call it the *dummy node vector* of T'_Z and denote it by B_D. The i-th bit is 1 if and only if the i-th parenthesis of U is '(' and its corresponding node is a real node in T'_Z. An example of a dummy node vector is also shown in Fig. 5. The 0-terminal node is also ignored. Let the FID of B_D be M. Using M, we can determine whether a node is dummy or real, and compute preorder ranks among only real nodes. We can also obtain positions of real nodes on BP from their preorder ranks by the select operation on M.

One-child array : An integer array to represent the 1-child of each node is called the *one-child array* and denoted by C_O. This array contains node preorder ranks of all 1-children in preorder on T_Z. That is, its i-th element is the preorder rank of the 1-child of the nonterminal node whose preorder rank is i. We also require one bit for each element of the one-child array to store the $\emptyset$-flag. If $\mathit{empflag}(v) = 1$ for a nonterminal node v, the corresponding element in the one-child array become negative. An example of a one-child array is shown in Fig. 6. Let I be the compressed representation of C_O. In I, one integer is represented by $\lceil \log(m+1) \rceil + 1$ bits, including one bit for the $\emptyset$-flag.

4 Algorithm

4.1 Convert algorithm

We show how to construct the DenseZDD. We first build the zero-edge tree from the given ZDD. A pseudo-code is given in Fig. 9 in the appendix. The zero-edge tree consists of all 0-edges of the ZDD, with their directions being reversed. For a nonterminal node v, we say that v is a 0^r-child of $\mathit{zero}(v)$. To make a zero-edge, we use a list *revzero* in each node, which stores 0^r-children of the node. The lists for all the nodes are computed by a depth-first traversal of the ZDD. This is done in $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ space, since each node is visited at most twice and the total size of *revzero* is the same as the number of nonterminal nodes.

We obtained a zero-edge tree T, but it is not an ordered tree. We define preorder rank $\mathit{prank}(v)$ for every node v before traversal. The nodes in *revzero* are sorted in descending order of their pairs $\langle \mathit{index}, \mathit{prank} \rangle$, that is, $\mathit{index}(\mathit{revzero}[i]) \geq \mathit{index}(\mathit{revzero}[i+1])$ for $1 \leq i < |\mathit{revzero}(v)|$. Then, nodes with higher indices are visited first. This ordering is useful to reduce the number of dummy nodes and to implement ZDD operations simply. It seems contradiction to define preorder rank of a node by preorder rank of its child, but it is possible. Since a ZDD node v satisfy $\mathit{index}(v) > \mathit{index}(\mathit{zero}(v))$ and $\mathit{index}(v) > \mathit{index}(\mathit{one}(v))$, we can decide *prank* for every node by the pseudo code in Fig. 7, which is a BFS algorithm based on *index* value starting from 0-terminal. To compute *prank* efficiently, we construct the temporary BP for the zero-edge tree. Using the BP, we can compute the size of each subtree rooted by v in T in constant time and compact space.

Next, we create dummy nodes imaginarily. For a node v, we create $q = \max\{\, i \in \{1, \ldots, n\} \mid i = \mathit{index}(\mathit{revzero}[j]) - 1, 1 \leq j \leq |\mathit{revzero}(v)| \,\}$ dummy nodes $d_1, \ldots, d_q$ such that $\mathit{triple}(d_i) = \langle \mathit{index}(v) + i, d_{i-1}, \mathbf{0} \rangle$, and $\mathit{empflag}(d_i) = 0, 1 \leq i \leq q$. For convention, d_0 denotes v.

The DenseZDD for the given ZDD is composed of these three data structure. We traverse the zero-edge tree in DFS order as if dummy nodes exist and construct the BP representation U, the dummy node vector M, and the one-child array I. The BP and dummy node vector are constructed for the zero-edge tree with dummy nodes. On the other hand, the one-child array ignores dummy nodes. Finally, DenseZDD $DZ = \langle U, M, I \rangle$ is obtained. Pseudo-codes are given in algorithms in Fig. 8, and 9 in the appendix.

4.2 Primitive ZDD operations

We show how to implement primitive ZDD operations on DenseZDD $DZ = \langle U, M, I \rangle$ except *getnode*. We give an algorithm for *getnode* in Section 5.

In the zero-edge tree, there are two types of nodes: real nodes and dummy nodes. Real nodes are those in the ZDD, while dummy nodes have no corresponding ZDD nodes. Real nodes are numbered from 1 to m based on preorders in the zero-edge tree. Below a node is identified with this number, which we call its *node number*. We can convert between the node number i of a node and the position p in the BP sequence U by $p := \mathit{select}_1(M, i)$ and $i := \mathit{rank}_1(M, p)$.

The 0-terminal has node number 0 and nonterminal nodes have positive node numbers. If a node number of a negative value is used, it means a node with an $\emptyset$-flag.

In addition, we consider an additional primitive operation for DenseZDDs: $\mathit{chkdum}(p)$. This operation checks if a node at position p on U is a dummy node or not. If it is a dummy *chkdum* returns false; otherwise it returns true. This operation is implemented by simply looking at the p-th bit of M. If the bit is 0, then the node is dummy; otherwise it is a real node.

$\mathit{index}(i)$: Since the item of the node is the same as the depth of the node, we can obtain $\mathit{index}(i) := \mathit{depth}(U, \mathit{select}_1(M, i))$.

$\mathit{one}(i)$: Because 1-children are stored in preorder of the parents of nodes, we can obtain $\mathit{one}(i) := I[i]$.

$\mathit{topset}(i, d)$: The node $\mathit{topset}(i, d)$ is the ancestor of node i in the zero-edge tree with index d. A naive solution is to iteratively climb up the zero-edge tree from node i until we reach a node with index d. However, as shown above, the index of a node is identical to its depth. By using the power of the succinct tree data structure, we can directly find the answer by $\mathit{topset}(i, d) := \mathit{rank}_1(M, \mathit{level\_ancestor}(U, \mathit{select}_1(M, i), d))$.

$\mathit{zero}(i)$: Implementing the *zero* operation requires a more complicated technique. Consider a subtree T of the zero-edge tree consisting of the node i, its real parent node r, all real children of r, and dummy nodes between those nodes. As a pre-condition, the zero-edge tree is constructed by Algorithm 9 in the appendix. That is, for the children of r, the nodes with higher *index* value have smaller preorder, and the imaginary parents of the children are dummy nodes (or i) that are added on the edge between r and the child having the highest *index* value. Computing $\mathit{zero}(i)$ is equivalent to finding r. Because the children of r are ordered from left to right in descending order of their depths, and dummy nodes are shared as much as possible, the deepest node in T is on the leftmost path from r. Furthermore, the parents of other real children are also on the leftmost path. This property also holds in the original zero-edge tree. The dummy node vector B_D stores flags in the preorder in the zero-edge tree. Then $B_D[p_r] = B_D[p_i] = 1$, where p_r and p_i are positions of nodes r and i in the BP sequence U, and $B_D[j] = 0$ for any $p_r < j < p_i$. Therefore we can find p_r by a rank operation. In summary, $\mathit{zero}(i) := \mathit{rank}_1(M, \mathit{parent}(U, \mathit{select}_1(M, i)))$.

4.3 Compressing the balanced parentheses sequence

The balanced parentheses sequence U is of length $2u$, where u is the number of ZDD nodes including dummy nodes. This number is equal to the size of the quasi-ZDD. If a ZDD has m real nodes and the number of items is n, the size u of its quasi-ZDD is mn in the worst case. Here we compress the BP sequence U.

The BP sequence U consists of at most $2m$ runs of identical symbols. To see this, consider the substring of U between the positions for two real nodes. There is a run '))...' followed by a run '(((...' in the substring. We encode lengths of those runs using some integer encoding scheme such as the delta-code or the gamma-code [3]. An integer $x > 0$ is encoded in $\mathcal{O}(\log x)$ bits. Because the maximum length of a run is n, U can be encoded in $\mathcal{O}(m \log n)$ bits. The range min-max tree of U has $2m/\log m$ leaves. Each leaf corresponds to a substring of U that contains $\log m$ runs. Then any tree operation can be done in $\mathcal{O}(\log m)$ time. The range min-max tree is stored in $\mathcal{O}(m(\log n + \log m)/\log m)$ bits.

We also compress the dummy node vector B_D. Because its length is $2u \leq 2mn$ and there are only m ones, it can be compressed in $m(2 + \log m) + o(u)$ bits by FID. The operations $select_1$ and $rank_1$ take constant time. We can reduce the term $o(u)$ to $o(m)$ by using a sparse array [14]. The operation $select_1$ is done in constant time, while $rank_1$ takes $\mathcal{O}(\log m)$ time.

From the discussions in this section, we can prove Theorem 1 and Theorem 2. For the proof, see the appendix.

5 Hybrid method

In this section, we show how to implement dynamic operations on DenseZDD. Namely, we need to implement the $getnode(i, v_0, v_1)$ operation. Our approach is to use a hybrid data structure using both the DenseZDD and a conventional dynamic ZDD. Assume that initially all the nodes are represented by a DenseZDD. Let m_0 be the number of initial nodes.

In a dynamic ZDD, the operation $getnode(i, v_0, v_1)$ is implemented by a hash table indexed with the triple $\langle i, v_0, v_1 \rangle$.

We show first how to check whether the node $v := getnode(i, v_0, v_1)$ already exists. That is, we want to find a node v such that $index(v) = i$, $zero(v) = v_0$, $one(v) = v_1$. If v does not exist, we create a such node by the hash table as well as a dynamic ZDD. If it exists, in the zero-edge tree, v is a real child node of v_0. Consider again the subtree of the zero-edge tree rooted at v_0 and having all real children of v_0. All children of v_0 with index i share the common (possible dummy) parent node, say w. Because w is on the leftmost path in the subtree, it is easy to find it. Namely, $w := level\_ancestor(U, select_1(M, rank_1(M, v_0)+1), i)$. The node v is a child of w with $one(v) = v_1$. Because all children of w are sorted in the order of *one* values by the construction algorithms, we can find v by a binary search. For this, we use *degree* and *child* operations on the zero-edge tree.

Theorem 3. *The existence of $getnode(i, v_0, v_1)$ can be checked in $\mathcal{O}(t \log m)$ time, where t is the time complexity of primitive ZDD operations.*

Table 2. Comparison of performance, where δ denotes the dummy node ratio

data set	#items	#nodes	#itemsets	size (bytes)			comp. ratio		δ
	Z			Z	DZ	DZ_c	DZ	DZ_c	
grid5	40	584	8512	17520	2350	2196	0.134	0.126	0.28
grid10	180	377107	4.1×10^{20}	11313210	1347941	1265773	0.119	0.112	0.20
grid15	420	1.5×10^{8}	2.3×10^{48}	4342789110	678164945	647843001	0.156	0.149	0.19
webview5	952	2299	11928	10592760	3871679	1851889	0.365	0.174	0.93
webview10	1617	6060	70713	281700	1034471	477299	0.367	0.169	0.93
webview20	2454	30413	634065	912390	290661	140873	0.318	0.154	0.92
webview50	2905	93900	4.4×10^{6}	181800	44846	24596	0.246	0.135	0.88
webview100	3149	353092	2.7×10^{7}	68970	11455	7967	0.166	0.115	0.75
webviewALL	3149	465449	3.2×10^{7}	13963470	4964303	2413625	0.355	0.172	0.92
randjoin128	32696	6751	2.5×10^{8}	202530	408149	99117	2.015	0.489	0.99
randjoin2048	32768	377492	1.8×10^{13}	11324760	2415648	1511658	0.213	0.133	0.82
randjoin8192	32768	1.3×10^{6}	3.7×10^{15}	38094930	5328502	4386452	0.139	0.115	0.42
randjoin16384	32768	1.9×10^{6}	2.8×10^{16}	56447280	7056418	6113910	0.125	0.108	0.14

If the BP sequence is not compressed, *getnode* takes $\mathcal{O}(\log m)$ time. Otherwise it takes $\mathcal{O}(\log^2 m)$ time.

6 Experimental results

We ran experiments to evaluate the compression, construction, and operation times of DenseZDDs. We implemented the algorithms described in Sec. 3 and 4 in C/C++ languages on top of the SAPPORO BDD package [9]. The package uses 30 bytes per ZDD node. We show the characteristics of the ZDDs in Table 2. The experiments are performed on a 3.09 GHz AMD Opteron with 512 GB memory running SuSE 10.

As real data sets, for $N = 5, 10, 20, 50, 100$, the source ZDD *webview*N was constructed from the data set BMS-Web-View-2[3] by using mining algorithm LCM over ZDD [12] with minimum support N. For artificial data sets, the ZDD *grid*N represents all self-avoiding paths on a $N \times N$ grid graph from the top left corner to the bottom right corner [5][4]. Finally, *randjoin*N is a ZDD that represents the join $C_1 \sqcup \cdots \sqcup C_4$ of four ZDDs for random families $C_1, \ldots, C_4$ consisting of N sets of size one drawn from the set of $n = 32768$ items.

In Table 2 we show the sizes of the original ZDD, the DenseZDD, and their compression ratio. We compressed FID for dummy node vector if the dummy node ratio is more than 75%. We observe that the DenseZDD is from five to ten times smaller than an original ZDD, and the compressed DenseZDD is half the size of the DenseZDD. For most of our data sets, the ratio δ of the number of dummy nodes to the size of DenseZDD is roughly 90%, except for *grid*N and *randjoin16384*.

[3] http://fimi.ua.ac.be

[4] An algorithm animation: http://www.youtube.com/watch?v=Q4gTV4r0zRs

Table 3. Converting time and random searching time

data set	conversion time (sec)				traverse time (sec)			search time (sec)		
	read	convert	const.	comp.	Z	DZ	DZ_c	Z	DZ	DZ_c
grid5	0.001	0.001	0.009	0.000	0.000	0.001	0.006	0.029	0.038	0.229
grid10	0.461	0.634	0.449	0.060	0.075	0.247	1.388	0.005	0.013	0.056
grid15	124.887	407.502	112.379	8.186	41.214	102.673	398.397	0.006	0.011	0.064
webview5	0.256	0.690	1.361	0.055	0.066	0.154	0.250	1.966	0.045	0.099
webview10	0.217	0.226	0.564	0.041	0.017	0.042	0.073	1.901	0.043	0.100
webview20	0.066	0.036	0.313	0.022	0.005	0.014	0.027	1.875	0.046	0.101
webview50	0.013	0.019	0.050	0.004	0.002	0.020	0.005	1.314	0.273	0.102
webview100	0.004	0.002	0.017	0.001	0.000	0.004	0.007	0.777	0.129	0.376
webviewALL	0.551	0.927	1.644	0.108	0.091	0.207	0.346	1.706	0.049	0.105
randjoin128	0.004	0.053	0.149	0.008	0.001	0.002	0.003	0.527	0.044	0.095
randjoin2048	0.243	0.742	0.946	0.029	0.093	0.126	0.145	8.071	0.044	0.098
randjoin8192	0.858	2.573	1.259	0.043	0.338	0.240	0.304	15.604	0.039	0.092
randjoin16384	1.270	5.016	1.471	0.070	0.676	0.353	0.447	19.501	0.040	0.093

In Table 3, we show the conversion times from ZDD to DenseZDD, traversal times, and search times on ZDD and DenseZDD. Conversion time is composed of four parts: time to read a file containing a stored ZDD and reconstruct the ZDD, convert it to raw parentheses, bits, and integers, construct succinct representation of them, and compress the BP of the zero-edge tree. The conversion time appears almost linear in the input size showing its scalability for large data. Traverse operation used $zero(v)$ and $one(v)$, while membership operation used $topset(v, i)$ and $one(v)$. We observed that the DenseZDD has almost twice longer traverse time and more than 10 times shorter search time than an original ZDD. These results show the efficiency of our implementation of the $topset(v, i)$ operation on DenseZDD using level ancestor operations.

From the above results, we conclude that DenseZDDs are more compact than ordinary ZDDs unless the dummy node ratio is extremely high, and the membership operations for DenseZDDs are much faster if the number of items is large or the dummy node ratio is small. We observed that in DenseZDDs, traversal time is approximately double and search time approximately one-tenth compared to the original ZDDs.

7 Conclusion

In this paper, we have presented a compressed index for static ZDDs named DenseZDD. We also proposed a hybrid method for dynamic operations on DenseZDD so that we can manipulate DenseZDD and conventional ZDD together. For future work, the one-child array should be stored in more compact space. We will implement the hybrid method on our ZDD package. We expect that our technique can be extended to other variants of BDDs.

References

1. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, and Shin-ichi Minato. Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations. In *Proceedings of the Prague Stringology Conference 2011 (PSC'11)*, pages 147–161, Czech Technical University in Prague, 2011.
3. Peter Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.
4. Esben Rune Hansen, S. Srinivasa Rao, and Peter Tiedemann. Compressing binary decision diagrams. In *Proceedings of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 799–800, 2008.
5. Donald E. Knuth. *The Art of Computer Programming, volume 4, fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley, 2009.
6. Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *Journal of Discrete Algorithms*, in press, January 2013.
7. P. Mateu-Villarroya and J. Prades-Nebot. Lossless image compression using ordered binary-decision diagrams. *Electronics Letters*, 37:162–163, 2001.
8. Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of DAC 1993*, pages 272–277, 1993.
9. Shin-ichi Minato. SAPPORO BDD package. Division of Computer Science, Hokkaido University, 2012. unreleased.
10. Shin-ichi Minato. Z-skip-links for fast traversal of zdds representing large-scale sparse datasets. In *Proceedings of ESA 2013*, volume LNCS 8125, pages 731–742. Springer, 2013.
11. Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of DAC 1990*, pages 52–57, 1990.
12. Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In *Proceedings of PAKDD 2008*, volume LNAI 5012, pages 234–246. Springer, 2008.
13. Gonzalo Navarro and Kunihiko Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 2012. Accepted. A preliminary version appeared in SODA 2010.
14. Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.
15. Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43:1–43:25, November 2007.
16. Mike Starkey and Randy Bryant. Using ordered binary-decision diagrams for compressing images and image sequences. Technical report, Carnegie Mellon University, CMU-CS-95-105, January 1995.

A Succinct data structures

A.1 Succinct data structures for rank/select

Let B be a binary vector of length u, that is, $B[i] \in \{0, 1\}$ for any $0 \leq i < u$. The rank value $rank_c(B, i)$ is defined as the number of c's in $B[0..i]$, and the select value $select_c(B, j)$ is the position of j-th c ($j \geq 1$) in B from the left. Note that $rank_c(B, select_c(B, j)) = j$ holds if $j \leq rank_c(B, n-1)$, the number of c's in B. The predecessor $pred_c(B, i)$ is defined as the position j of the rightmost $c = B[j]$ to the left of $B[i]$. The predecessor is computed by $pred_c(B, i) := select_c(B, rank_c(B, i))$.

The Fully Indexable Dictionary (FID) is a data structure for computing rank and select on binary vectors [15].

Theorem 4 (Raman et al. [15]). *For a binary vector of length u with n ones, its Fully Indexable Dictionary uses $\binom{u}{n} + \mathcal{O}(u \log\log u / \log u)$ bits of space and computes $rank_c(B, i)$ and $select_c(B, i)$ in constant time on the $\Omega(\log u)$-bit word RAM.*

This data structure uses asymptotically optimal space because any data structure for storing the vector uses $\lceil \binom{u}{n} \rceil$ bits in the worst case. Such a data structure is called a *succinct data structure*.

A.2 Succinct data structures for trees

An ordered tree is a rooted unlabeled tree such that children of each node have some order. A succinct data structure for an ordered tree with n nodes uses $2n + o(n)$ bits of space and supports various operations on the tree such as finding the parent or i-th child, computing the depth or the preorder of a node, etc., in constant time [13]. An ordered tree with n nodes is represented by a string of length $2n$ called a balanced parentheses sequence (BP), defined by a depth-first traversal of the tree. Starting from the root, we write an open parenthesis '(' if we arrive at a node from above, and a close parenthesis ')' if we leave from a node upward.

In this paper, the following operations are used. Let P denote the BP sequence of a tree. A node is identified with the position of the open parenthesis in P representing the node.

- $depth(P, i)$: the depth of a node at position i. (The depth of a root is 0.)
- $preorder(P, i)$: the preorder of a node at position i.
- $level\_ancestor(P, i, d)$: the position of the ancestor with depth d of node i.
- $parent(P, i)$: the position of the parent of node i (identical to $level\_ancestor(P, i, depth(P, i) - 1)$).
- $degree(P, i)$: the number of children of node i.
- $child(P, i, d)$: the d-th child of node i.

The operations take constant time.

A brief overview of the data structure is the following. The BP sequence is partitioned into equal-length blocks. The blocks are stored in leaves of a rooted tree called range min-max tree. In each leaf of the range min-max tree, we store the maximum and the minimum values of node depths in the corresponding block. In each internal node, we store the maximum and the minimum of values stored in children of the node. By using this range min-max tree, all tree operations are implemented efficiently.

B Pseudo codes

Global vaiables: $L_1, \ldots, L_n$ are list which are empty initially.
ALGORITHM Compute_Preorder (L_0)
Input: L_0: a list stores only $\langle\{\mathbf{0}\}, [0, \mathit{stsize}(\mathbf{0}) - 1]\rangle$;

```
1: for i = 0, ..., n
2:    for each ⟨A, [l, r]⟩ ∈ L_i in arbitrary order % A is a set of nodes
3:       for each v ∈ A in descending order of ⟨prank(one(v)), empflag(v)⟩
4:          prank(v) ← l++;
5:          for each j ∈ { j | w ∈ revzero(v), j = index(w) } in descending
            order
6:             A ← { w | w ∈ revzero(v), index(w) = j };
7:             r ← l + sum{ stsize(w) | w ∈ B };
8:             append ⟨B, [l, r]⟩ to L_j;
               % That is, the prank of descendants of nodes in B are in [l, r].
9:             l ← r + 1;
10: return;
```

Fig. 7. An algorithm which computes the preorder rank $\mathit{prank}(v)$ of each node v. Sets of nodes are implemented by arrays or lists in this code. The $\mathit{prank}(\mathbf{0})$ is 0.

C Proof

Following is the proof for Theorem 1 and 2.

Proof. We first prove Theorem. 1. From the above discussion, the BP U of zero-edge tree costs $2u = \mathcal{O}(mn)$ bits where u is the size of corresponding quasi-ZDD. The one-child array needs $m \log m$ bits for 1-children and m bits for $\emptyset$-flags. Using FID, the dummy node vector is stored in $m(2 + \log m) + o(u)$ bits. Therefore, the DenseZDD can be stored in $2u + m \log m + 3m + o(u)$ bits and primitive operations except *getnode* are done in constant time because the rank_1, select_1,

ALGORITHM Convert_ZDD_BitVectors $(v, paren, dummy, onechild)$
Input: ZDD node v, list of parentheses $paren$,
list of bits $dummy$, list of integers $onechild$

```
 1: i = index(v);
 2: for each w ∈ revzero(v) in ascending order of prank(w);
 3:    while i + 1 < index(w)
 4:       append '(' to paren, and '0' to dummy;
 5:       ++i;
 6:    append '(' to paren, and '1' to dummy;
 7:    append prank(one(w)) · (−1 · empflag(w)) to onechild;
 8:    Convert_ZDD_BitVectors(w, paren, dummy, onechild);
 9:    append ')' to paren, and '0' to dummy;
10: while i > index(v)
11:    append ')' to paren, and '0' to dummy;
12:    −−i;
13: return;
```

Fig. 8. Algorithm for obtaining the BP representation of the zero-edge tree, the dummy node vector, and the one-child array.

and any tree operations take constant time. Since the *getnode* finds a target node by binary search, it takes $\mathcal{O}(\log m)$ (described in Sec. 5).

Next, we prove Theorem 2. When we compress U, it can be stored in $\mathcal{O}(m \log n)$ bits and the min-max tree is stored in $\mathcal{O}(m(\log n + \log m)/\log m)$ bits. The dummy node vector can be compressed in $m(2+\log m)+o(m)$ bits by FID with sparse array. But, the time order of any tree operations and the $rank_1$ operation is changed from constant time to $\mathcal{O}(\log m)$ time. Therefore, the DenseZDD can be stored in $\mathcal{O}(m(\log m + \log n))$ bits and primitive operations take $O(\log m)$ times larger than the above time because all of them use tree operations or $rank_1$ on M.

ALGORITHM Construct_DenseZDD (W: list of ZDD root)
Output: DenseZDD DZ

1: **for each** $v \in V$ compute *revzero* fields for all descendants of v;
2: compute *stsize* field for all 0^r-decsendants;
3: **Compute_Preorder**($\{\langle\{\mathbf{0}\}, [0, stsize(\mathbf{0}) - 1]\rangle\}$);
4: create empty lists $paren, dummy, onechild$;
5: append '(' to *paren*, and '0' to *dummy*;
6: **Convert_ZDD_BitVectors**(**0**, paren, dummy, onechild);
7: append ')' to *paren*, and '0' to *dummy*;
8: make BP U from *paren*;
9: make FID M from *dummy*;
10: make compressed representation I of *onechild*;
11: **return** $DZ \leftarrow \langle U, M, I\rangle$;

Fig. 9. Algorithm for constructing the DenseZDD from a source ZDD.